Software Tools for Practical Work with Formal Task Descriptions: A Case Study with an Extended GOMS Technique[1]

Thomas Strothotte, Berlin
Peter Fach, Berlin
Erik Olsson, Uppsala
Lars Reichert, Heidelberg

Abstract
Formal task descriptions are of theoretic interest but have yet to achieve their breakthrough in practical software development. We conjecture that this lack of their use in practice stems to a large extent from the facts that (1) they are tedious and difficult to write, (2) there are no procedures for checking their correctness and (3) they find almost no uses other than for descriptive purposes. For these reasons, they are presently not economical for use in real software development. In this paper we motivate and describe the design and implementation of a set of tools in the form of dialog systems for alleviating these problems. The complete package, which allows practical work with an extended GOMS technique, consists of about 10.000 lines of C and Presentation Manager code running on an IBM PS/2 under OS/2.

1. Introduction

The quality of a user interface depends not only on its functionality but to an equal extent also on the tasks which users are to carry out with the software. The tasks and the interface must make a good match; checking this the primary job of a usability tester. Formal task analyses are one promising tool for guiding systematic studies of the usability of a software product.

Numerous methodologies for carrying out task analyses and for formally describing tasks have been developed over the last decade. Ideally, an appropriate analysis should be carried out early in the software development process so that it can affect the design to help make the user interface more appropriate for solving the problems which users will actually want to solve. However, the realities of task analyses in practical software development processes are quite different: Typically no formal analyses are carried out at all and even if then only as a **post mortem** when preparing a usability test.

---

[1] This work was carried out while the authors were at the IBM Scientific Center, Heidelberg.

We feel that this lack of practical use of formal task descriptions finds its roots in the following aspects:

- **Writing formal task descriptions is difficult.**
  A great deal of time and effort is typically required to produce a formal task description, since many details must be specified. Formal methods enforce rigor and completeness, hence parts which are considered unimportant must be specified in as much detail as parts which are deemed to be crucial.

- **It is difficult to check a formal description.**
  How can one be sure that the description actually corresponds to what its author wanted to describe? Unlike with programming languages, for example, task descriptions are not directly executable.

- **Formal task descriptions tend not to be re-used for other purposes.**
  With only a small number of recent exceptions, task descriptions are used for the sake of description only. Little effort has been expended on using them for the purposes of designing or constructing something else. Hence the effort to construct a task description is often not economical.

Indeed, we conjecture that these problems are not unique to formal task descriptions but actually common to many formal methods. For example, denotational semantics [Tennant 1981] is a theoretically sound method of defining the semantics of computer programs but is virtually unused in software houses.

In this paper we report on steps toward alleviating these fundamental problems with formal task descriptions. Our aim is to reduce the effort required to produce formal task descriptions while allowing more to be done with them so as to make their use more economical. We assume an environment in which a usability tester is given a prototype of a software product and wishes to work with a formal task description as a first step in his usability evaluation. We designed and implemented a prototypical workbench which is to help the usability tester in his work with one of the formal task description methods, in our case GOMS* which was developed by U. Arend [Arend 1989] [Arend 1990] in cooperation with the IBM Heidelberg Scientific Center, and which we extended further. The workbench consists of several interactive dialog systems on an IBM PS/2 under OS/2 with the Presentation Manager. We tested our systems with a prototypical application.

The paper is organized as follows. Chapter 2 briefly describes formal task descriptions and their role in usability testing. The description method used in the present study, an extension of GOMS*, is outlined. The design of our workbench discussed in Chapter 3. In Chapter 4, some lessons learned during the implementation are discussed. Concluding remarks are made in Chapter 5.

2. Formal Task Descriptions

The formal specification of tasks has been a topic of interest within computer science since the late 1970's. The goal is to be able to analyze in a systematic manner the expected performance and competence of users of a software system. Such specifications provide a basis for stating hypotheses about the human-computer interaction which can be tested through experimental sessions of users working with the system. Furthermore, they provide a basis for explaining experimental observations.

Numerous formalisms have been presented in the literature: a BNF-oriented approach [Reisner 1977] [Reisner 1981], the keystroke-level model [Card/Moran/Newell 1983], GOMS (standing for Goals, Operators, Methods and Selection Rules) [Card/Moran/Newell 1983], an extension of GOMS called GOMS* [Arend 1990], Cognitive Complexity Theory [Kieras/Poulson 1985], Task-Action Grammars [Payne/Greene 1986], Extended Task Action Grammars [Tauber 1990], and notations for help-systems [Schwab 1988] [Hoppe 1988].

Most of the above mentioned task description methods address direct manipulative interfaces, which are of particular interest in the context of the present study. The approach taken is to reduce the description of the interaction down to a command-language "equivalent" (for example, dragging the icon for a file X on top of the trash can is treated like a command "delete X"). Only one of the techniques [Tauber 1990] attempts to incorporate the spatial relationships among the objects; however, his scheme is rather complicated and has not been tested in practice. [Arend 1989] also bases his work on a command-oriented representation, but suggests ways of extending it to model visualizations and spatial relationships.

While each of the above-mentioned techniques has various strengths and weaknesses, none can be considered to be the all-purpose solution, in particular when dealing with user interfaces with direct manipulation. Indeed, we conjecture that all the methods have the drawbacks alluded to in the introduction. To work on these problems, it was thus prudent to chose one method and work with it. The GOMS* method appeared most appropriate as a basis.

GOMS* is an extension of GOMS. Like GOMS, its basis is formed by the specification of user goals, operators (motoric actions carried out by the user), methods which the users invokes to achieve his goals and selection rules for deciding which of competing methods to use. The main new aspect of GOMS* is that goals and operations are treated more precisely through the introduction of new predicates in the formalism. Further, a number of Algol-like control structures (REPEAT, WHILE etc.) are introduced to make the flow of control explicit. While these additions make the task descriptions longer and more complicated, they help in providing more details for the specification of tasks.

Arend is able to use GOMS* for analyzing the consistency of the user interface (similar to TAG) and he is able to predict the time-to-learn for a software system. Further, he suggests that automatic logfile analysis could be carried out with GOMS* and that user errors can also be analyzed. Finally, he discusses how his work can be extended so as to integrate aspects of the screen layout. Such extensions are necessary to pay tribute to the unique requirements of

direct manipulative interfaces.[2] An example of part of a GOMS*-like description is illustrated in Figure 1.

```
Method( Entryfield, type_in )
     if (Entryfield.Text == FALSE )
       . G: SPECIFY( Entryfield )
       . . O: MOVE( Mouse, Entryfield )
       . . O: CLICK( Right_button)
       while (Entryfield.Text == FALSE)
         . G: SPECIFY( Key )
         . . O: PRESS( Key )
       endwhile
     endif
```

Figure 1. Example of a GOMS*-like description: This method describes a subtask for text input into an entry-field.

3. Design and Implementation of a Task Description Workbench

Recall that we are assuming an environment in which a user is presented with a software product and wishes to construct and work with a GOMS-like task description. In this chapter we describe the facets of a software workbench to facilitate practical work with our extended GOMS*. The workbench has three main modules, corresponding to the three deficits described in the introduction. In the first module, a new concept for the construction of a task description is realized. The second allows checking the correctness of a task description in a new way. With the third, a task description can be exploited to guide the compilation of end-user documentation for the software product.

3.1 Constructing Task Descriptions

3.1.1 Design

The intuition behind our method of constructing task descriptions stems from the observation that they typically contain a significant amount of regularity. There is much repetition, often with only small variations.

We are able to exploit this regularity. Our strategy is to allow the author to demonstrate to the system the tasks "by example" and having the system construct the formal description from

---

[2] In fact, we have made such extensions [Reichert 1990] and the implementations we describe in the remainder of the paper work with them. However, these extensions alter nothing in principle for the present study and are beyond the scope of this paper.

the example. After globally planning the tasks, the author carries them out with the product (see Figure 2).

All the author's actions are monitored by an "observation" program running in a second window; periodically he switches to it to specify certain structural information of the task, such as the names of high-level goals. As a result, the observation program is able to construct a GOMS*-like description of the the tasks demonstrated to it.

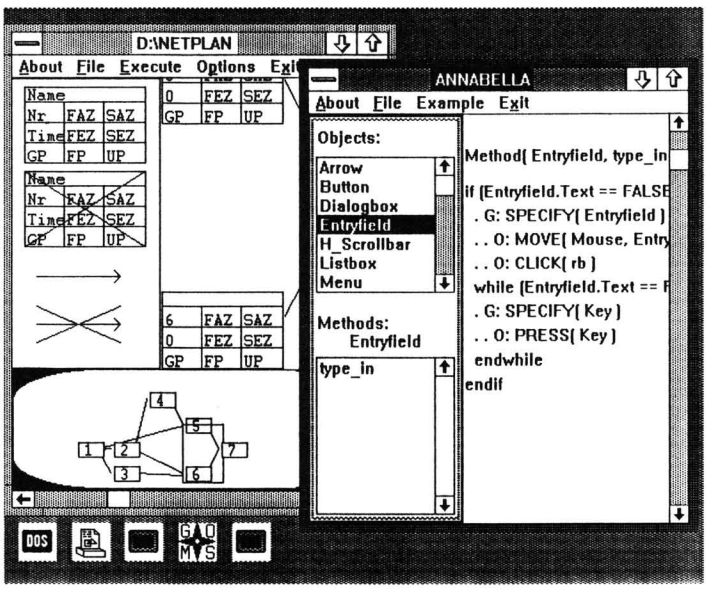

Figure 2. Screen layout for dialog system for inputting task descriptions "by example": The author carries out the tasks with the software product (here a graphical activity network editor, running in the left window). Periodically he comments, in a menu-based dialog with the observation program (right), the demonstration. As a result, the information necessary for a complete task description is gathered and its formal description constructed (extreme right).

3.1.2 Implementation

A prototype corresponding to the description above has been implemented [Reichert 1990]. The observation program has predefined GOMS*-like "subtask schema" associate with each of the object types which the Presentation Manager can produce such as windows, menus and dialog boxes. Furthermore, the user can extend this set of schema by performing examples and naming them in the observation program. A task description is constructed by the user performing the tasks, which are reduced down to their constituent subtasks by the observation program.

The Presentation Manager proved to be a very good environment for the purposes of an observation program. An application programmer is forced to make explicit many details of his program, such as the names of menus and their selection items and details of windows in a standardized manner. Moreover, at run-time this information is available to external processes, as is information pertaining to what the user is doing with respect to these items. Thus much of the low-level information necessary for a GOMS* task description can be collected by the observation program. Furthermore, a structure of the lower levels of the description can be ascertained by window- and menu-manipulations of the user.

3.2 Checking Task Descriptions

3.2.1 Design

Irrespective of how the task description was constructed - whether with the tool just described, with another tool or "by hand" - we wish to make sure that it corresponds to the software product with which users can carry out the tasks. Our technique is to design a kind of compiler for the task description (see Figure 3).

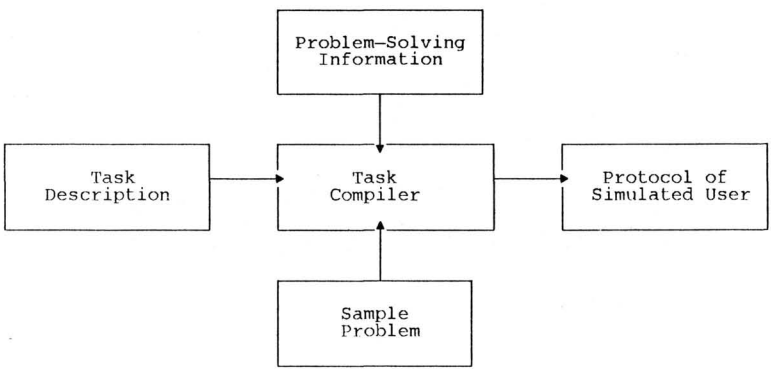

Figure 3. Checking task descriptions: The diagram shows the components of the system. A "task compiler" accepts as input a task description, a sample problem and problem-solving information. The output is a protocol of a simulated user solving the problem with the software product. This can then be piped into the application, allowing the person wishing to check the task description to observe the solution to the sample problem.

The input to the compiler is a task description, a sample problem and problem-solving information. The output is a sequence of keystrokes, like a logfile, simulating an end-user solving the sample problem with the product. This protocol is then piped into the application and the author of the task description can sit back and watch the problem being solved on the screen in front of him. While this cannot be a proof of correctness of the task description, just as

successful sample runs of a program do not yield a proof of correctness of the program, it nonetheless can give the author a feeling for the appropriateness of his task description.

3.2.2 Implementation

A detailed design of a system corresponding to the above architecture has been worked out and parts of an implementation in Prolo and C have been completed [Olsson 1990]. The problem-solving component essentially finds a path in the task description which when followed will solve the given problem. This implies that the problem-solving component must keep track of state information which is missing in GOMS-like task descriptions. A further difficult aspect of the problem-solving component turned out to be that the application is object-oriented while GOMS* is not.

The protocol produced by the compiler is written into a file. In a second pass, a resident program first loads the application, reads the protocol file and by calling up appropriate interrupts leads the application to think that a real user is entering data, whereas in fact the input is coming from the file via the resident program.

The protocol file also contains information as to the goals which the user is pursuing during his work. These goals, obtained directly from the task description, accompany the simulation of the end-user. Thus the author of the task description can not only follow what the simulated user is doing but also what he is "thinking", i.e., what goals he has set for himself and what methods he is using to achieve them. The person checking the description can informally convince himself of the appropriateness of these goals and thus the task description.

3.3 Using Task Descriptions to Design End-User Documentation

3.3.1 Design

End-user documentation is an aspect of software which is gaining in importance. Since user interfaces have traditionally been command-language oriented, end-user documentation has centered around the definitions of commands. This is not appropriate for user interface with direct manipulation because of their emphasis on "languageless" input. Furthermore, special care must be taken to convey to the reader the feeling for spatial relationships and synchronously presented information on the screen.

We conjectured that end-user documentation for user interfaces with direct manipulation could be drastically improved by organizing it in a task-oriented manner. In our prototypical system, the person compiling the documentation is led through tasks and can make screen-dumps upon demand. He then adds documentation to the screen dumps in the form of graphic symbols and usually some textual explanations. Thus the documentation focuses on the visual impressions of the interface. Our work builds on that of [Gong/Elkerton 1990] which instead focuses on the sequential verbal aspects of the tasks and their underlying concepts.

The design of the system is based on the "dual coding" theory of human information processing [Paivio 1986] which we applied to the problem of designing end-user documentation. The theoretic background is beyond the scope of this paper; the interested reader is referred to [Fach 1990].

3.3.2 Implementation

The system which we have implemented is independent of the application and runs on the PS/2 as a separate process parallel to the application. It consists of the following modules:

- **Window-Inspection module**: Upon demand, this generates bitmap representations of the window contents at run-time.
- **Picture-Editor**: The person compiling the documentation can edit the bitmaps and assure their correspondence to subtasks. The editing can go so far as to construct comic-strip-like sequences of documentation.
- **Run-time system for end-users**: These are programs for providing end-users wishing to use the documentation access to the information. This access is either via the picture (bitmap) or via keywords. Access via motoric actions is in principle also possible, though this aspect has not yet been implemented.

The end-user thus has on-line documentation which is picture-oriented and centered around tasks. In principle, hard-copy documentation could be produced but such a module has not yet been implemented.

4. Discussion

We tested the tools described in Chapter 3 with a prototypical application. For this purpose, we designed and implemented a program to construct activity networks [Elmaghraby 1977], also using the IBM Presentation Manager. This implementation, as well as that of the tools, turned out to be significantly more effort than initially estimated. The Presentation Manager provides a solid basis for programming high-quality user interfaces but is difficult to manage; it is akin to writing assembler code. However, this kind of programming, in which all kinds of information has to be coded explicitly in data structures which are made available to other processes, is ideal for the kind of tools which we designed. Practical work with the tool for constructing task descriptions showed that this kind of interaction is much more pleasant for the author than using pencil and paper to write the description. Further, our experience shows that the technique is less prone to errors.

The component to check task descriptions is plagued by a number of fundamental problems. First, if the task description is constructed with the tool described in this paper, then it will already correspond to the application and hence the checking procedure is not likely to be needed. Second, since the checking is accomplished essentially by a person watching the sim-

ulation of a user, errors in the granularity of the task description well go unnoticed, particularly if the person checking the simulation is also the author of the task description. Finally, it turned out to be a significant effort to write the problem-solving component; since this must be done for each application, this process should be supported by appropriate tools.

The methodology for compiling end-user documentation we have developed takes steps toward bridging the gap between a verbal explanation of an action and actually carrying it out. Since the emphasis is on the pictorial impression of the screen, we feel that the user can better recognize practical situations during his work and be able to recall what to do based on the documentation through which he previously worked. Here we are able to tap in on the enormous human memory capacity for pictures.

5. Concluding Remarks

We have shown that working with formal task descriptions can be simplified with some appropriate tools. However, our work is only the "tip of the iceberg"; there is significantly more room for improvement in this direction.

In our work we assumed the practical situation of today, that is that a formal task analysis is carried out when a prototype of a software system is ready for usability evaluation. Of course, this analysis should be carried out earlier in the design process. This, in turn, implies that the task description should be less bound to th actual interaction style of the software. Indeed, GOMS and its derivative we used describe the tasks down to the keystroke level, which is actually too detailed. The description should be cut off at a conceptually higher level and combined with a description of a potential user interface for the tasks. Usability information could then be ascertained from the mapping of the tasks onto the interface characteristics. Since the trend in user interface development is going more and more in the direction of object-oriented programming, it would make sense to use a more strongly object-oriented task description method [Reichert 1990].

A new and potentially important use formal task descriptions is in the area of rapid prototyping. To improve usability testing, it would be useful to have a prototype of a software product produced automatically or at least semi-automatically from a task description. This way, potential usability problems could be tested much earlier and much more simply.

6. Acknowledgements

The authors wish to thank Udo Arend for many helpful discussions.

7. References

[Arend 1989] U. Arend, "Analysing complex tasks with an extended GOMS Model", in: D. Ackermann, M. Tauber (Eds.), *Mental Models and Human-Computer Interaction*, North-Holland, Amsterdam, 1989.

[Arend 1990] U. Arend, *Wissenserwerb und Problemlösen bei der Mensch-Computer-Interaktion*, Dissertation, Universität Darmstadt; S. Roderer Verlag, Regensburg, 1990.

[Card/Moran/Newell 1983] S. K. Card, T. P. Moran, A. Newell, *The psychology of human-computer interaction*, Lawrence Erlbaum, Hillsdale, NJ, 1983.

[Elmaghraby 1977] A. Elmaghraby, *Activity Networks*, Academic Press, New York, 1977.

[Fach 1990] P. Fach, *Aufgabenorientierte Endbenutzer-Dokumentation für Benutzerschnittstellen mit direkter Manipulation*, Diplomarbeit, Fakultät Informatik der Universität Stuttgart, August, 1990 (written at IBM Heidelberg).

[Gong/Elkerton 1990] R. Gong, J. Elkerton, "Designing minimal documentation using a GOMS model: a usability evaluation of an enginnering approach", in: J. C. Chew, J. Whiteside (Eds.), *Human Factors in Computing Systems. Proc. CHI '90*, ACM/SIGCHI, New York, 1990, pp. 99-106.

[Hoppe 1988] H. U. Hoppe, "Task-oriented parsing - a diagnostic method to be used by adaptive systems", in: E. Soloway, D. Freye, S. B. Sheppard (Eds.), *Human Factors in Computing Systems, Proc. CHI '88*, ACM/SIGCHI, New York, 1988, pp. 241-247.

[Kieras/Polson 1985] D. E. Kieras, P. G. Polson, "An Approach to the Formal Analysis of User Complexity", *International Journal of Man-Machine Studies 22* (1985), pp. 365-394.

[Olsson 1990] E. Olsson, *Generierung aus formalen Aufgabenbeschreibungen*, Examensarbete, Faculty of Mathematics, University of Uppsala, Sweden, September, 1990 (written at IBM Heidelberg).

[Paivio 1986] A. Paivio, *Mental Representations: A Dual Coding Approach*, Oxford University Press, Oxford, 1986.

[Payne/Green 1986] S. J. Payne, T. R. G. Green, "Task-action grammars: a model of the mental representation of task languages", *Human-Computer Interaction 2* (1986), 93-133.

[Reichert 1990] L. Reichert, *Formale Beschreibungen von direktmanipulativen Benutzerschnittstellen*, Diplomarbeit, Fakultät Informatik der Universität Stuttgart, September, 1990 (written at IBM Heidelberg).

[Reisner 1977] P. Reisner, "Use of psychological experimentation as an aid to development of a query language", *IEEE Trans. on Software Engineering 3*(3) (1977), pp. 218-229.

[Reisner 1981] P. Reisner, "Formal grammar and human factors design of an interactive graphic system", *IEEE Trans. on Software Engineering 7*(2) (1981), pp. 229-240.

[Tauber 1990] M. Tauber, "ETAG - Extented Task Action Grammar - A Language for the Description of the User's Task Language", in: D. Diaper et al. (Eds.), *Human-Computer Interaction - INTERACT '90*, Elsevier Science Publishers (North-Holland), pp. 169-174.

[Tennant 1981] R. D. Tennant, *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

Authors

Prof. Dr. Thomas Strothotte, Institut für Informatik, Freie Universität Berlin, Nestorstraße 8-9, D-1000 Berlin 31, Germany.
Dipl.-Inform. Peter Fach, Institut für Informatik, Freie Universität Berlin, Nestorstraße 8-9, D-1000 Berlin 31, Germany.
Erik Olsson, Faculty of Mathematics, University of Uppsala, Uppsala, Sweden.
Dipl.-Inform. Lars Reichert, IBM Wissenschaftliches Zentrum Heidelberg, Tiergartenstraße 15, D-6900 Heidelberg, Germany.