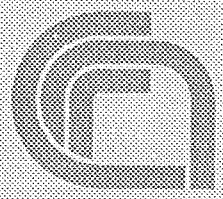
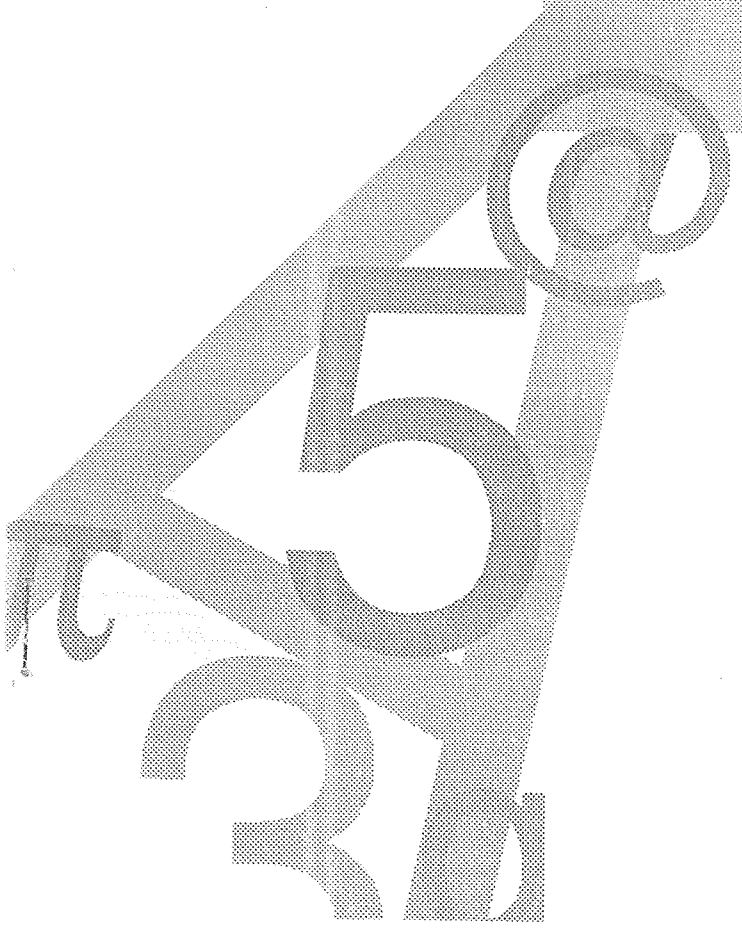


Consiglio Nazionale delle Ricerche

User Interface Evaluation Using Task Models

A. Lecerof e F. Paternò

CNUCE: C03/97



User Interface Evaluation Using Task Models

A.Lecerof (+, *), F.Paternò (+)

(+) CNUCE-CNR - Via S.Maria 35 56126 Pisa

(*) Department of Computer and Information Science, Linkoping University

{andreas, fabio}@giove.cnuce.cnr.it

Italy

Abstract

The main goal of this work is to propose a method to evaluate user interfaces using task models and logs from a user test of an application. The results of the method give the designer information about how to improve the user interface. These results include an analysis of the tasks which have been accomplished and those which failed, and of user errors and their type. This allows the evaluator to see if the specified usability goals were satisfied. The results from the evaluation also include time related information, task patterns among the accomplished tasks and the available tasks from the current state of the user session.

Index Terms: User Interfaces, Formal Methods for Human-Computer Interaction, Usability Engineering, Task Models, User Interface Evaluation, Dialogue Models

1. Introduction

The interest in human factors connected to the development of interactive software applications has increased considerably over the last few years. A key concept in the Human Computer Interaction (HCI) field is *usability*, which is concerned with making systems easy to learn and easy to use. An important step towards the goal of usability is the evaluation of the user interface. The results from the evaluation should provide information to the designer about how to improve the user interface and thus reach higher goals of usability.

Usability engineering is an area in HCI which aims to achieve usable systems by different methods. This paper concerns the evaluation part of usability engineering. With the use of a task model and the logs from a user test of the application considered, we propose a method to evaluate a user interface. The information from the analysis should then be used to improve the usability of the application.

Evaluation is a central part of systems development. This is due to the support it gives to the design team through the design process by giving them feedback on how they can improve their work. At early stages of the design process evaluation is used to sort out alternative designs and to predict different aspects of the design. Later on, evaluation is performed to measure if the design meets its requirements.

Before an evaluation is performed it is important to know what the goal of the evaluation is. The main purpose for performing an evaluation is usually one of the following (Preece et al., 1994):

- Engineering towards a target: is the design good enough?

- Comparing alternative designs: which is the best?
- Understanding the real world: how well does the design work in the real world?
- Checking conformance to a standard: does this product conform to the standard?

User interface evaluation is often performed through techniques that require user interface expertise. In *heuristic evaluation* (Nielsen, 1993) user interface specialists study the interface and look for properties that they know, from experience, lead to usability problems.

Another approach is *usability testing* (Dumas & Redish, 1994). In this case the designer studies the user performing some tasks and gather data on the problems that arise. The problems with usability testing are that it is based on only observational data and to be able to interpret the data, still some user interface experience is needed. There is also the problem of cost, you take up the time of the users, and the problem of getting the appropriate users for the current test.

Alternative methods have been proposed, they try to structure the evaluation process so that designers, the interface developers, can carry out the evaluation. One example is the use of *guidelines*, which provide the evaluators with specific recommendations about the design of the interface, such as how the contents of a screen should be organized or how items should be arranged (see e.g. Brown, 1988).

The method we propose is also intended to be carried out by designers. We try to combine usability testing with the information from the task model of the application. The aim is to give the designer the necessary information about how to improve the interface.

The use of task models can give a more user-centred base for design of user interfaces (Diaper 1989). This is because task models do not only allow designers to specify the functionalities of the user interface but also the users accesses to these functionalities and the related interactions.

The process of task analysis usually begins with investigating the users' current problem. This is done by breaking down the tasks that potential users of the system do or would do, into sequences of actions and objects. We define a task model as the description (often hierarchically structured) of the set of activities necessary to reach a goal. The goal is a state of the system the user wishes to achieve.

In usability engineering task models have been used to develop interactive applications. Tools such as TLIM (Paterno' et al., 1996) and Adept (Wilson et al., 1994) use task models to support user interface design. The idea is to create the user interface from a declarative specification, that gives an abstract description of the user interface.

The usual application of task models is thus in the development and design of computer systems. In this work we will use task models differently, because our goal is to evaluate a user interface, not to design one.

When we use the task model we will use it for *describing* the interaction techniques supported by existing user interfaces too. This means we will refine the tasks specification in more detail. The task model gives us the temporal relationships among tasks. With this information it is possible to see, at every state of the user session, the tasks that are possible to perform.

An attempt to use task models in evaluation can be found in (Hamilton, 1996). The aim is to evaluate the usability early in the design phase at the conceptual level, to predict the difficulties users might have. This is done by using the task knowledge the users have when achieving their goals. Our approach differs though because we want to evaluate an existing user interface later in the design

process when the task model is established. We also use the task model differently, e.g. to get the temporal relationships among tasks.

AIDE (Sears 1995) is a tool that uses "simple task descriptions" to guide the design and evaluation of an interface. The aim of the tool is to assist designers in creating and evaluating layouts (Sears, 1993) for a given set of interface widgets. It is based on five metrics; efficiency, alignment, horizontal balance, vertical balance and constraints. The *efficiency* evaluates how far the user must move a cursor to accomplish a task. In contrary to our approach, AIDE does not involve user testing, but is based merely on metrics. We are more concerned about the users' tasks and how the users achieve to perform their tasks, and not only the efficiency of the layout.

In (Paterno et al., 1995) they evaluate a user interface using the architectural specification and the task model of the user interface. Our approach is similar but we omit the architectural specification and use a more advanced task model, allowing designers to express a richer set of relationships among tasks. This gives more useful information to understand why the errors occurred during the user test. The reasons for the errors can guide the designer how the user interface can be improved.

More specifically, the purpose of this work is:

- *To find a method to evaluate a user interface using task models and logs from a user test of an application*

  After an evaluation the evaluator should have measurable quantities, like the number of tasks and errors performed, making it possible to see if the usability goals set up were satisfied. The results should also give the designer necessary information about how to improve the interface.

- *To develop a tool that uses the method above, with the purpose to facilitate and give automatic support to the evaluation process.*

  The tool should support the evaluator when applying the method. It should also give the results from the evaluation in different formats, allowing the user of the tool various possibilities to choose how s/he wants to present the results.

The paper is structured in the following way: Part 2 gives an introduction to usability engineering and user interface evaluation, describes three evaluation approaches and discusses a comparison among them. Part 3 explains the task model used in this work and how it can be used for user interface evaluation. Part 4 describes the proposed method for evaluation and its results. It also gives an example of use. Part 5 describes the tools used in this work, the Replay tool to record the user logs and the USINE tool to execute the evaluation method. Part 6 gives an example of an application evaluated according to the proposed method. The user interface and the task model of the application are explained, then the results from the evaluation are described and discussed. Finally we draw some conclusions from this work and give some suggestions on future work.

2. Usability engineering

Basic Concepts in Usability Engineering

Usability engineering is an approach to system design where the usability of a product is specified in quantitative terms and in advance. If you are developing a system according to usability engineering you must therefore have measurable usability goals. This makes it possible to decide if a finished product reaches the usability goals set up by the designer or the design team. The underlying goal for usability engineering is to improve. Not only to improve the user interface but also to improve the first steps of the design process, e.g. by letting users participate in the early stages.

3

To understand what *usability engineering* is about it is important to first know what usability means. Usability is concerned with making systems easy to use and easy to learn. But usability includes more than just "easy to use" and "easy to learn". For example, if the users can not carry out all the tasks they want, because a feature is missing in the system, it is not likely that they will agree that the system is usable. A broader definition is therefore to prefer, a definition that at least includes the following:

- The *relevance* of the system, how well it serves the users' needs.
- The *efficiency*, how efficiently users can carry out their tasks using the system.
- The users' *attitude* to the system, their subjective feelings.
- The *learnability* of the system, how easy the system is to learn for initial use and how well the users remember how to use the system.
- The *safety* of the system, giving the users right to "undo" actions and not allowing the system to act in a destructive way, e.g. to delete files without telling the user.

The most important part of usability depends though on the actual system. For some programs, e.g. for children, it is very important that they are easy to use, for other systems, such as banking systems, efficiency is one of the important parts.

In usability engineering you try to operationalize "your" definition of usability for the current system to make measurable goals. This means that you have to define usability in terms of measurable quantities, e.g.:

- User performance on specified tasks, measured in terms of task completion rate, completion time or number of errors;
- Users' subjective preference or degree of satisfaction;
- Learnability, measured in task completion rate, completion time, number of errors, or use of documentation and help desk.
- Flexibility, how well the system can change when the requirements change.

These quantities and similar can be the result of an evaluation of a user interface. The benefits of having measurable terms are among other things:

- It is likely that the resources put into designing for usability will increase. This means, that if you have measurable goals of usability that you must reach, it is likely that you will work to fulfil these goals.
- You will know how much further work is required on usability.
- It will be easier to compare requirements of alternative designs.

User Interface Evaluation

User interface evaluation is a process to gather data about the usability of the design of a user interface. A reason for doing an evaluation could be that you want to know if the current design meets the requirements in the requirement specification. Another reason could be that you want to compare two different designs.

The important thing to know when evaluating a user interface is what usability means for the current application. For example, a flight control program must have certain features that a word processor must not have. This is why an analysis of users and their needs is important. If we do not know what the user wants and needs, we can not know which tasks s/he must be able to perform.

4

Evaluation can be performed at different times in the development process. During the early stages evaluations tend to be done to predict the usability of the product or to check the design team's understanding of the users' requirements. Later on in the design process the focus is more to identify usability problems and to improve the user interface. This is the focus we use in this work.

An evaluation of a user interface after performing a user test should include the following:

- *The tasks the user succeeded to perform*

  When testing the user interface the user should be given different tasks s/he should perform. If the user succeeded to perform the tasks it is likely that the interface is usable. But there is also a risk that the tasks you specified were too easy.

- *The tasks the user did not succeed to perform*

  These tasks are important because they indicate problems the user had with the user interface. It is likely that the user needs some help to perform these tasks.

- *How many times each task was performed*

  If a task is performed frequently it could be implemented as a shortcut or a macro.

- *In which order the tasks were performed*

  The designer often has an opinion in which order the user will perform the tasks. If the user wants to break this order it could signify that the designer must make it possible to perform the tasks in a different order. Another aspect concerns if the user always chooses to perform one task before another. Then the second task could be executed automatically when the first one is performed.

- *The different errors the user did*

  The user errors can be of different types. If the user makes an action not necessary to perform the current task this is, in most cases, an error. However, there is also the possibility that the user wanted to go backwards in the interaction, to a previous step. The reason for this could be the user wants to be sure s/he is on the right way before continuing. For example, before the user decides to delete some selected files s/he may want to control that s/he has made a correct selection, i.e. so only the files meant to be deleted are selected. Another type of error is an action performed belonging to the task, but the user has missed to do some actions needed before. For example, if the user tries to print out a file without specifying the name of the file first. In this case the user tried to do the right thing (to push the print-button) but the *precondition* of the task (to specify the name of the file) was not satisfied. An error could also arise if the user gives input to the program that is not correct. For example, if the user is trying to divide a number by zero. Then the user lacks information about the current domain.

- *The user's subjective opinion of the user interface*

  The user's opinion is important because if the user does not like the interface, it is not likely that s/he will use it. The user can give useful information to the designer on which part of the user interface s/he liked and disliked. The quality of this information depends though on the user's knowledge of computer science and experience of user interfaces.

- *How much help the user needed*

  If the user needed a lot of help it is possible that the user interface was too difficult, i.e., this is a measure of how easy the user interface was.

5

- *How many times the user had to restart from the beginning*

  This measures how difficult it was to navigate in the program.

- *How long time it took the user to perform the tasks*

  If the time is an important factor of the specification of the software, which often is the case, then it is meaningful to measure the time.

Based on the information above the designer should suggest improvements that could be made to the user interface.

Related Works

The *MUSiC performance measurement method* (Macleod et al., 1994) was developed by the European MUSiC (Metrics for Usability Standards in Computing) project to provide valid and reliable means of specifying and measuring usability. The method gives feedback useful how to improve the usability of the design. MUSiC also includes tools and techniques for measuring user performance and satisfaction. Products supporting the method are available, e.g. DRUM (Macleod & Rengger, 1993), a video analysis software.

The basic outputs of the MUSiC performance measurement method include measures of: *Effectiveness*, how correctly and completely goals are achieved in a certain context; *Efficiency*, effectiveness related to cost of performance (calculated as effectiveness divided by the time). Together with the DRUM tool, which supports the analysis of a video recording of a usability test, the full (video supported) method also includes the following measures and diagnostic data: *Relative User Efficiency*, an indicator of learnability, how easy the system was to learn, relating the efficiency of users to that of experts. *Productive Period*, the proportion of time the user spent not having problems. *Snag, Search and Help times*, time spent overcoming problems, searching unproductively through a system, and seeking help. These measures are valuable data about specific areas where the design fails to support the users' performance. The method provides suggestions of causes of the problems. A benefit of these quantitative data is that they enable a comparison of alternative designs. The diagnostic information of the full method also gives help identifying where improvements of the user interface have to be made.

To measure the user satisfaction the method includes the use of a questionnaire, SUMI (Kirakowski & Corbett, 1993). The satisfaction is said to be fulfilled if: the user feels that the system supports the way they carry out their tasks; the user feels in command of the system; the system is helpful and easy to learn.

To provide measures of goal achievement, the MUSiC method uses two concepts, quantity and quality: *Quantity*, the proportion of the task goals represented in the output of a task which have been attempted; *Quality*, the degree to which the task goals represented in the output have been achieved

To evaluate performance, tasks must be chosen so that they have an outcome that can be assessed. For example, when evaluating a query database the user wants to get specific items of information. The output is then analysed to measure how many items the user retrieved, and how well all the retrieved items met the original requirements.

The effectiveness with which the user performs a task, is defined as a function of the *quantity* of the task the user completes, and the *quality* of the goals s/he achieves. The quality and the quantity are subjective criteria decided before the analysis, by a person familiar with the current task. After the effectiveness is calculated it is also possible to decide the *efficiency* of the task performance. The

6

efficiency is defined as the effectiveness divided by the task time. The efficiency is useful when you want to compare similar products or the performance of the same task at different occasions.

Unlike the MUSiC method, *cognitive walkthrough* (Wharton et al., 1994; Lewis & Rieman, 1993) can be done very early in the design phase. Cognitive walkthrough is an evaluation method mainly focused on how easy a system is to learn, especially through exploration. Users often prefer to learn a program by exploration instead of taking a course or reading the manual. The users tend to first learn how to perform the tasks important for their work and then learn new features when needed.

The idea of cognitive walkthrough comes from "code walkthrough" used in software engineering. In code walkthrough the sequence of code is stepped through to find bugs and to check the quality of the code. The aim of cognitive walkthrough is to get the users' thoughts and actions when using an interface for the first time. A brief description includes the following. First of all it is necessary to have a prototype or a detailed design of the interface as well as facts about the users, who they are and which tasks they have in their work. Then choose a relevant task that the design is intended to support. After this you try to tell a believable story about each step and action the user has to do to fulfil the task. To make the story believable you have to motivate every action the user does, based on knowledge about the user and the feedback the user gets from the system. If you can not tell a believable story about an action you have found a problem with the interface.

A cognitive walkthrough can find different problems with a user interface. It can find out how a user will think, e.g., why should the user think that the printer had to be turned on? It can identify controls, obvious to the designer, but hard to find for the user, and it can note difficulties with labels and bad feedback (what is on and off?). The focus of the cognitive walkthrough is on the problems users have when they use a system for the first time. For some systems "walk-up-and-use" system this is specially interesting. Examples are banking machines and tourist information systems, which often are used without any training. But more complicated systems are also interesting, users often learn them incrementally, starting as beginners and learning new functions when needed.

It is important to remember that cognitive walkthrough is a method to develop and improve a user interface not to validate it. This means that the purpose of a cognitive walkthrough is not to see if a user interface fulfils its requirements in the requirements specification. The walkthrough should instead be done with the aim to find things that could be improved with the interface.

Before performing a cognitive walkthrough you must have the four following things: a description or a prototype of the user interface; a textual informal description of one of the tasks the user should be able to perform with the program; a *complete* list of the actions needed to fulfil the task; knowledge about the user and his/her work tasks.

During the cognitive walkthrough you try to tell a believable story why the user chooses to perform an action. Meanwhile you criticise the story to be sure it is believable. The following four questions are used to criticise the story: 1) Will the user try to perform the actions needed for the task? 2) Will the user find the control (the menu, the button, etc.) for the current action? 3) After the user has found the control, will s/he understand that it performs what the user wants? 4) After the action is performed, will the user understand the feedback s/he gets so s/he can go on to the next action with confidence?

The first question deals with how the user will think, because users often do not think as the designer expected. The second question is about if the users will find the existing controls and understand

7

which actions are possible. This can be a problem with some designs, e.g., remote controls where some buttons are hidden for design reasons. The third question deals with if the user will understand the current control. (What will happen if you push the button? Will the printer be switched on or off?) The last question is about the feedback the user gets. Will the user understand the messages the system gives or the meaning of a highlighted icon?

When the cognitive walkthrough is finished you try to deal with the found problems. Some things are obvious, use understandable labels and give better feedback. The advantage with a cognitive walkthrough is that the result includes how the users think when they use a system, their goals and their assumptions. Cognitive walkthrough easily finds problems with missing or difficult labels and if the feedback is insufficient.

(Jeffries et al., 1992) compared four evaluation techniques, heuristic evaluation, software guidelines, usability testing and cognitive walkthrough. They found that cognitive walkthrough misses general and recurring problems. For walk-up-and-use systems, where it is specially important that the user quickly and easily can use and understand the interface, cognitive walkthrough still remains as a good alternative. One must remember though that cognitive walkthrough does not test real users on the system. Therefore you must be aware of that it is unlikely that you will find all the existing problems of the user interface with the method.

Empirical testing, e.g. testing used in the MUSiC method, consists of iterative testing and design revision where real users test and help to find usability problems. The major disadvantage with this method is that is expensive. This is because you use the time of the users, thus preventing them for carrying out their *real* work. It is also a problem to find necessary numbers of users who are both domain experts and who belong to the target group. Because of this, "discount methods" (Nielsen 1994) and "inspection methods" (e.g. cognitive walkthrough) have been developed, for assessing the usability of an interface design more quickly and at a lower cost. The lower cost is due to that you do not use real users in the methods. Another possibility is given by *engineering models* of usability. The aim is to produce *quantitative predictions* of how well users will be able to perform tasks with a proposed design. The goal is also to capture the essence of the design in an inspectable representation. Usually the designer starts with an initial task analysis and a proposed first interface design. The designer should then use an engineering model (like GOMS, Card et al. 1983) to find the applicable usability problems of the interface. A GOMS model is a representation of the *procedural knowledge* the user must have in order to carry out tasks, their "how to do it" knowledge.

One of the purposes of GOMS is to *predict* the *execution time* which is simulated by executing the actions required to perform the tasks. Each action is divided into smaller parts until the remaining action is a simple keystroke or a mouse click. The times for all actions are then counted and summed up into a prediction of the time it will take to perform the whole task. The detail of the evaluation is very extreme, down to keystrokes, therefore it is also called the "keystroke-level analysis". The method makes it possible to predict the time to complete a task within a 20 percent margin (Lewis & Rieman, 1993).

To use the GOMS model you must conduct a task analysis to identify what goals the user will try to accomplish. The designer can then express in the GOMS model how the user can accomplish these goals with the system being developed. The task analysis however does not replace the understanding of the user's situation, working context and goals. The GOMS model can only predict the *procedural* aspects of usability, as the amount, consistency and efficiency of the procedures that users perform. Since many systems depend heavily on the simplicity and efficiency of the procedures,

8

the GOMS model can be useful. The reason why GOMS can predict these aspects of usability is because the methods for accomplishing user goals tend to be tightly constrained by the design of the interface.

Recent work as GLEAN (Kieras et al., 1995) has the purpose to allow interface designers or analysts to easily develop and rapidly apply GOMS model techniques in order to evaluate a user interface. The GLEAN user (the interface designer) will develop a GOMS model for an existing or proposed interface and supply a representative task and a description of the interface behaviour. GLEAN will then simulate the user interaction and generate usability metrics such as the learning time for the task. The most important property of GLEAN is that it automates the tedious calculations required to generate usability predictions from the GOMS model. The GLEAN tool also supports the reuse of GOMS methods, making it possible to use the same GOMS methods with a later revised version of the interface. However, the designer still needs to perform a task analysis to determine what goals the user is trying to accomplish. Another advantage of GLEAN is that it makes the GOMS model notation more readable and easier to understand.

The main aim of GLEAN is to simulate the interaction between a simulated user who interacts with a simulated device (the interface) to execute a set of benchmark tasks. To set up the simulation, the designer makes three descriptions: the "Task Instance Descriptions" to specify the task; the description of the user's procedural knowledge, represented with the GOMS model; the description of the behaviour of the interface, specified by the "Device Behavior Description", which specifies the objects in the interface, e.g. icons on the screen. The GLEAN tool then generates measures of usability from the GOMS model, such as predicted learning and execution time.

The study of Kieras et al. shows that GOMS together with automated tools like GLEAN can give an efficient usability evaluation, specially if more than one revision of the interface is done. Where the time is a critical part of the properties of the system this method can be of much benefit. However, the user of GLEAN must still perform a GOMS model. To capture all of the aspects of usability some form of user testing is still required to ensure a quality result.

The MUSiC method and usability testing are preferable used when the cost is not the critical part. It differs from the other methods that it is quite straightforward to use (relatively to the other methods). With this we mean that to perform an adequate usability test a long experience and education is not required (though it helps). The major advantage though is that it involves *real* users testing the user interface. The involvement of real users makes it more expensive but at the same time more reliable, at least from the users' point of view.

Cognitive walkthrough is known to be a long process both in preparation and performance time when used for larger systems. It has the advantage though that it concerns the users' thoughts and assumptions and that it can be used by the software developers. In walk-up-and-use systems, where learnability (the time to learn the interface) is an important aspect, cognitive walkthrough can be of much advantage.

Engineering models such as GLEAN are methods best used early in the design to predict the execution time. It is quite hard to use for non experts and takes long time regarding to the result it gives. There is hope though that future versions of GLEAN can reduce the evaluation time. Another disadvantage of engineering models like GLEAN, is that they presume an error-free behaviour of the user when performing the task. This means that you do not take user errors into consideration.

9

The aim of the method we propose is to overcome some of the limitations of the methods described in this part: it is based on the task model of the user interface and logs from a user test of the application considered. The tasks and the errors the user performs are the central parts. The aim of the proposed method is that an analysis of the tasks and the errors will give suggestions on how to improve the interface.

The disadvantage (and one of the limitations) of the MUSiC method is that the users' different tasks are not taken under major consideration. The measures from MUSiC are good if you want to compare different interface alternatives, but they are hard to use in order to improve the user interface, which is the purpose of our evaluation method.

Cognitive walkthrough and GLEAN does not involve directly users in the evaluation. We believe that involving users can give better information about how to improve the usability of the user interface. Unlike GLEAN our approach takes into account also the user errors, because we think they can give important information, guiding the designer how to improve the interface. We also want our method to be easy, so that the designer does not have to spend the major part of the time on evaluating, but on improving the user interface.

The proposed method should unlike GLEAN and cognitive walkthrough be used later in the design, to evaluate an existing user interface.

3. Task models for usability engineering

In this part we describe the task model we use in this work. We also discuss how it can be used for user interface evaluation.

Task models in usability engineering have been recognised as an important contribution to develop usable systems (Diaper 1989). Tools such as TLIM (Paterno' et al., 1997) and Adept (Wilson et al., 1994) use task models to support user interface design. With a task model we mean the description of the performance of the possible tasks. The notation for the task model we are using is expressed by *ConcurTaskTrees* (Paterno' et al., 1997). This notation allows designers to specify the temporal relationships among tasks and other information, useful for the design of the user interface.

The *ConcurTaskTrees* notation is used in the TLIM (Task LOTOS Interactor Modeling) method (Paterno' et al., 1996). This method allows designers to design and develop a user interface, starting with a task model of the application.

The TLIM environment includes a graphical editor (see Figure 1) for creating the task tree and specifying relationships among tasks. Only the parts of the TLIM method interesting for this work are described here. For example, the TLIM method also includes a transformation, from the task model to an architectural model of the system being developed, that is not described below (for more information, see Paterno' et al., 1996).

10

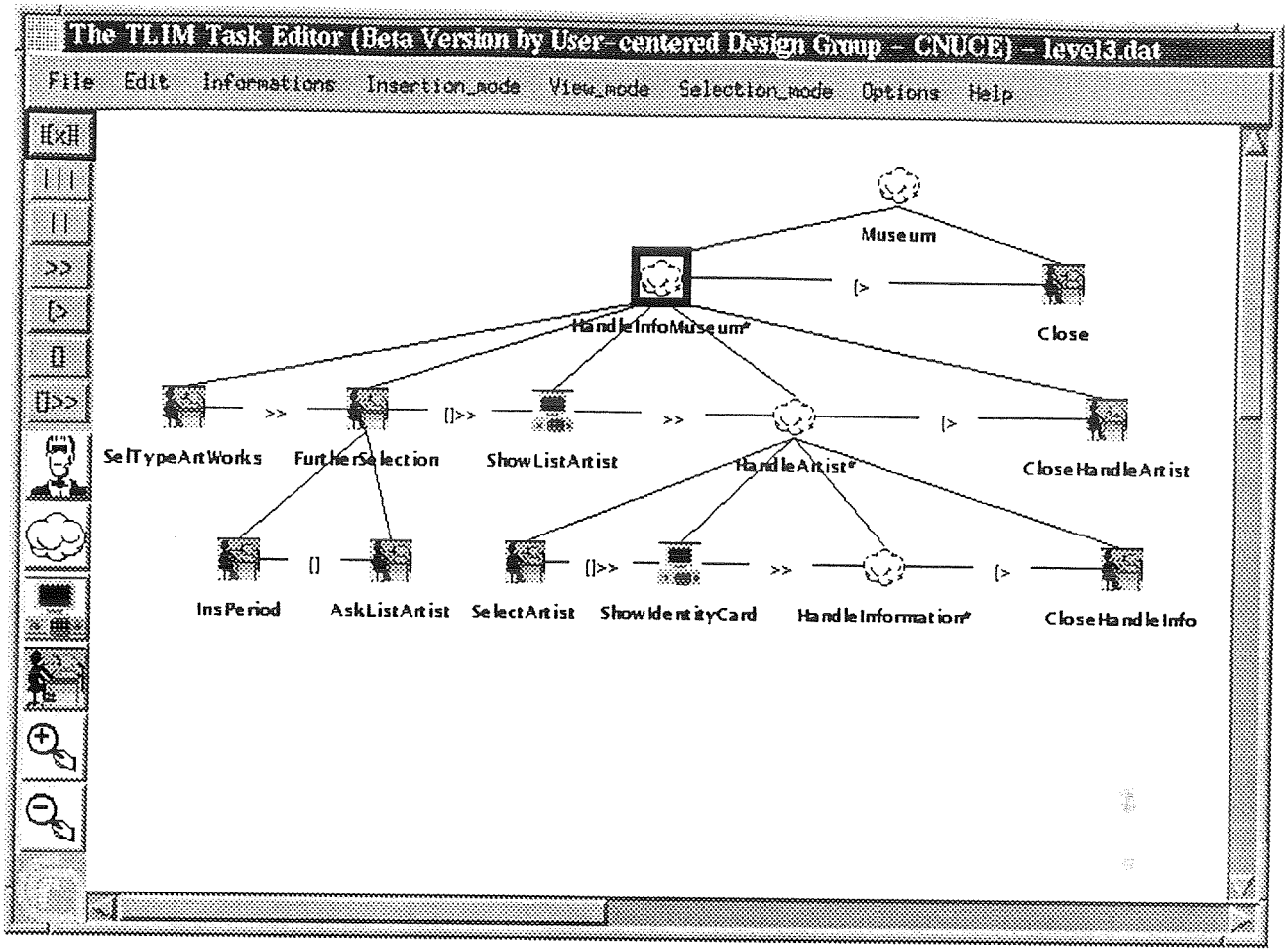


Figure 1 The TLIM-editor

The ConcurTaskTrees notation

We define a task as the set of activities required to reach a goal. The goal is a state of a system the user wishes to achieve. In the TLIM model we have four types of tasks (see Figure 2):

- *User tasks,* tasks performed by the user, e.g., to read a message. The user tasks are performed without any interaction with the system.
- *Application tasks,* tasks completely executed by the system, e.g., results given from a database. Application tasks receive information from the system but they can also supply information to the user.
- *Interaction tasks,* tasks performed by the user interacting with the system, e.g., pushing a button.
- *Abstract tasks,* tasks which require complex actions whose performance allocation has not yet decided, e.g., a user session with a system.

11

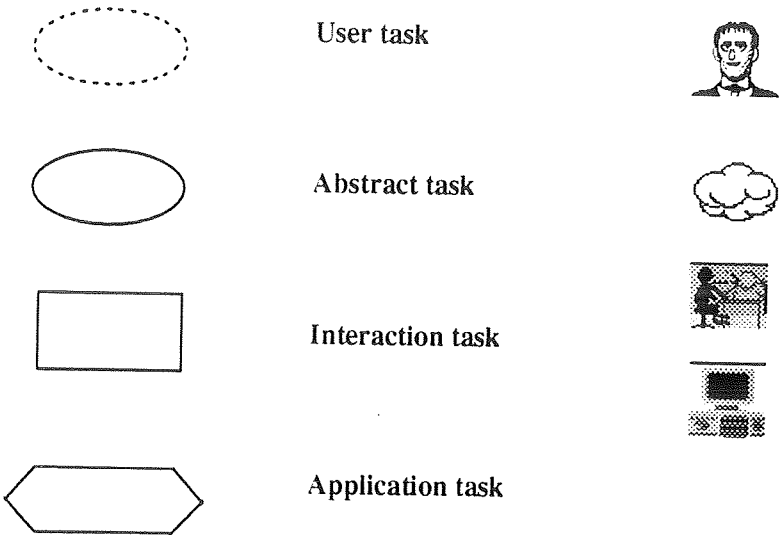


Figure 2 Different graphic presentations of task types

The task model is built in three phases:

- Firstly, we make a hierarchical and logical decomposition of the tasks in a tree structure. High level tasks (the abstract tasks) are decomposed into subtasks.
- Secondly, we identify the temporal relationships between the tasks.
- Finally, we identify the objects associated with each task. Objects are things we need and use to perform the task, e.g., an object of the task "send a letter" is the letter itself.

Objects and Actions

The objects associated with each task are of two kinds, perceivable and internal. *Perceivable objects* are items that users can interact with using their senses, e.g., menus, windows, icons and sounds. They can only belong to application or interaction tasks. *Internal objects* belong to the application and need help from the perceivable objects to be presented to the user. Examples are a query to the database and the data of the database.

Actions are simple tasks that do not involve any problem solving or thought. In the TLIM method actions are associated with objects, and can be of three different kinds; cognitive, logical or physical. For every task we specify what the last and the first action is.

The task can thus be described by the following attributes:

- Name
- Type of task; abstract, user, application or interaction.
- Subtask of; the name of the father task.
- Objects; the name, the type (internal, perceivable), a list of input actions, a list of output actions.
- Iterative; a boolean indicating whether the task is iterative.
- First action; the possible initial action.
- Last action; the possible final action.

12

The temporal relationships among tasks

The temporal relationships between the tasks are expressed by extending operators of the LOTOS notation (ISO 1988) which is a concurrent notation. It allows us to describe concurrent tasks unlike GOMS (Card et al., 1983) which only can analyse sequential tasks.

The operators we use are:

- T1 ||| T2, *interleaving*, task T1 and task T2 can be performed in any order.
- T1 |[]| T2, *synchronisation*, the two tasks T1 and T2 have to synchronise, i.e., to be executed at the same time, to exchange information.
- T1 >> T2, *enabling*, the performance of task T1 makes it possible to perform T2. Task T2 is not possible to perform until T1 is terminated.
- T1 []>> T2, *enabling with information passing*, when task T1 terminates it gives information to T2 besides activating it.
- T1 [] T2, *choice*, at the start both tasks are available but when one of them is started, the other is no longer available.
- T1 [> T2, *deactivation*, when task T2 is performed it is not longer possible to perform T1.
- T1*, *iteration*, the task is iterative and can be performed more than one time.

We can also specify *optional tasks*, tasks the user decides if s/he wants to perform or not. An example is default values, such as the number of copies that are to be printed, usually set to one. If the user is satisfied with the default value, s/he does not have to do anything. Optional tasks are marked within brackets "[]" e.g., [Choose nr of copies].

A problem when building task models using these operators is the possibility of ambiguity of expressions. For example, in Figure 3, we can interpret the specification in two ways:

(T1 [] T2) ||| T3 or T1 [] (T2 ||| T3).

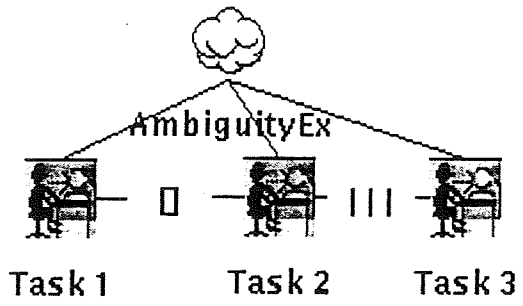


Figure 3 An example of ambiguity

To solve the ambiguity we have two possibilities. The first possibility is to use the priority order among operators defined by the standard LOTOS:

choice > parallel composition > disabling > enabling

If the designer does not want to use this priority another possibility is to introduce a task (Task D) which disambiguates the expression, see Figure 4.

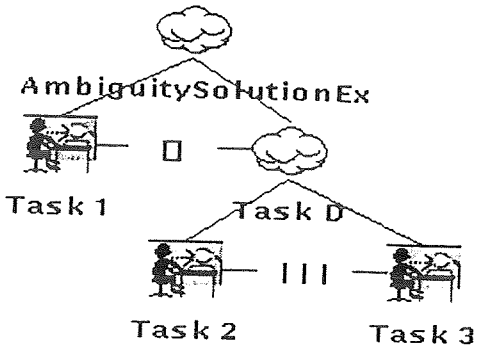


Figure 4 A solution to solve ambiguity

An example of a task tree using the TLIM method

In the tree in Figure 5 we can see that the *Cancel* task disables the *PrintSession* task and the whole left part of the task tree. This means that after we have performed *Cancel* we cannot perform anything more. The *Choose file* task enables the task *Print*. This means that the task *Print* gets necessary information from the task *Choose file* to be available.

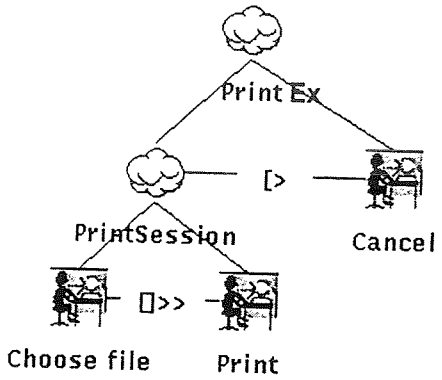


Figure 5 A small example of a task tree

How the task model can be used for evaluation

Task models are often used in the development and design of computer systems. In this work we use it for a different purpose though, to evaluate a user interface. This means that we will use tasks *differently*. For example, it is only the *interaction* tasks that the user really can perform during a user test. This means that our interest is concentrated to those tasks. However, we use other types of tasks as well, e.g., *abstract* task to specify complex tasks. When we use the task model we will also specify the tasks in more detail, i.e., for every user action corresponding to a log in the log file, we will have a basic task in the task tree. More complex tasks, this is still possible by using the abstract tasks.

The advantage of the task model is that it gives at every state the tasks possible to perform. This is due to the description of the temporal relationships among tasks. For example, look at task *PrintSession* in Figure 5 above which is connected to *Cancel* with the disabling operator. If *Cancel* is performed we will know that the task *PrintSession* and its entire subtree is impossible to do anymore, because task *Cancel* disables task *PrintSession*. Another example is the task *Choose file* above, it is enabling *Print*. Thus we know that *Choose file* must be done before it is possible to perform *Print*. Task *Choose file* is the precondition to the task *Print*. If task *Choose file* is done then and only then can we perform *Print*.

14

When the user performs a task s/he will change the current state of the task tree. By navigating through the task tree and indicating the tasks the user has performed, it is possible to tell which tasks at each state the user can perform. These tasks, *the available tasks*, can be found due to the temporal relationships among tasks.

The available tasks can be used for evaluation purposes to find the errors performed by the user. An error occurs when the user tries to perform a task not allowed according to the state of the task tree, e.g. when the precondition of a task is not satisfied.

The information about the state of the task tree is not only useful for evaluation of a user interface but also for generating task-oriented contextual help (Pangoli & Paterno', 1995).

In the evaluation of the user interface we examine the tasks the user tries to perform. If the user tries to perform a task which is not allowed in the current state of the task tree, we will record this as an error. The reasons for the errors differ. For example, it could be because the precondition of the task the user tried to perform was not satisfied or that the task was already disabled by another task.

4. The proposed method

In this paragraph we describe the proposed method for evaluating a user interface and the results achieved.

The input

Figure 6 below shows the structure of the method. The designer creates the task model with the TLIM editor (see part 3) and the logs tasks table with the USINE tool (see part 5). The user logs are provided by the log recording tool, Replay, from a user test of the current application.

The precondition table is then created automatically from the task model. The evaluation method finds the available tasks at the current state of the user session and provides the results from the evaluation.

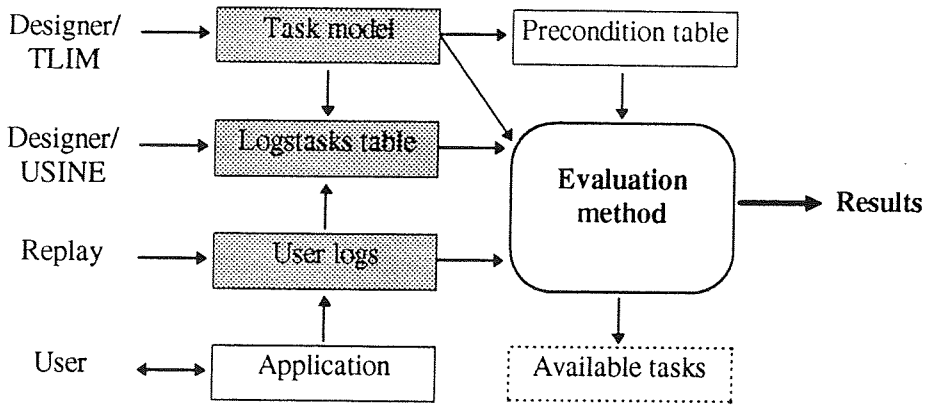


Figure 6 Overview of the method

That is, as input our method requires four things:

15

- Logs from a user test of the application considered
- The task model of the application
- The log-task table, created with information from the two above
- The precondition table, automatically created from the task model

The logs

We will get the logs from the user test by recording the user actions with the Replay tool (see part 5). The logs are saved in a simple text file. Before we can use the logs we have to clean them from redundant information.

The task model

The task model, as described in part 3, is created by using the TLIM-editor (http://verdolo.cnuce.cnr.it/task.tgz). We use the task model to be able to specify the temporal relationships among tasks, i.e., the enabling and disabling tasks. The task tree is done so detailed that every basic task is completed by one action (one log) in the logfile.

The log-task table

In the log-task table logs and tasks are associated by using the USINE tool (see part 5). That is, on one line we will find a log and the corresponding basic task the log executes. The log-task-table thus consists of two columns (see Figure 7).

The first column contains a **log** from the logs file. The logs specified in the log-task table consist of the *type* of action and the *target* of the action. For example the type can be a mouse click and the target the name of the widget, e.g. a list.

The second column contains the name of the corresponding **task** that the log *executes*. That is, when the log is performed the task is accomplished. Thus with each task we associate a corresponding log. If the task is not an interaction task we put a "#" instead of the log. This is because we can only record the user actions really occurred and not what the application has done internally.

```
text {/Comments}     |     Give Comments
click {/Vote}        |     Send vote
click {/Premiers}    |     Premiers
#                    |     Session
```

Figure 7 Examples from a log-task table; logs to the left, tasks to the right

In the figure above "click {/Vote}" executes the task "Send vote". The task "Session" is an abstract task, indicated with a "#".

The precondition table

In the task model of the application it is possible to see the tasks enabling and disabling other tasks (see part 3). From the task model we automatically create a table of preconditions, the *precond-table*, used internally in the method. In the precond-table we have the name of the task and the precondition or preconditions (if more than one) that must be performed before the task can be

16

completed. When the user performs a precondition we change the name of the precondition to the label *true* indicating that the precondition has been performed. Thus the notation used is:

<task>|<precondition>|...

Or, if the precondition was performed:

<task>|true|...

For example, the task *Send vote* has one task as precondition, *Write your name*. In the precond-table we would write:

```
Send vote|Write your name
```

If the user performs the task *Write your name* the field is changed to *true*:

```
Send vote|true
```

If the task has more preconditions than one, we write them after each other separated with a "|". For example, the task *Select/insert data* (an abstract task) has the following tasks as preconditions, *Choose cinema, Write a name* and *Write phonenr*. In the precond-table we would thus write:

```
Select/insert data|Choose cinema|Write a name|Write phonenr
```

This is because three tasks must be completed before the task *Select/insert data* itself can be accomplished. If we performed the task *Write a name* the precond-table would change to the following:

```
Select/insert data|Choose cinema|true|Write phonenr
```

The method for getting the preconditions from the task model

The method for finding the preconditions and creating the precond-table searches through the task tree top-down, left to right. For every task we check if its ***left brother is an enabling task***. If this is the case we know that the left brother is the precondition to the current task and write this (the task and its precondition) to the result.

There is also the possibility that the left brother is an abstract task. This is the case when we want to define more than one precondition to a task. Then we must first specify an abstract task as precondition to the current task, and then search for the preconditions for the abstract task, which are among the children of the abstract task.

During the search of the task tree the method collects the results. The method is thus as described above divided into two parts:

- If the current task has a left enabling brother, the left brother is the precondition of the current task.

  For example see Figure 8, if the current task is *Login* we will find that it has a left enabling brother, *WritePassword*. This means that *WritePassword* is the precondition to *Login*.

- If the left brother of the current task is abstract we must check the children of this task to get the preconditions.

  This is the case of the task *PersonInfo* in Figure 8 below. The current task is *LoginDialog* with the abstract left enabling brother *PersonInfo*. We will first put *PersonInfo* as the precondition to *LoginDialog*. To get the preconditions of the abstract task *PersonInfo* we must then check the children of this task. We will find that both children are preconditions of *PersonInfo*, namely *Write name* and *Write phonenr*.

17

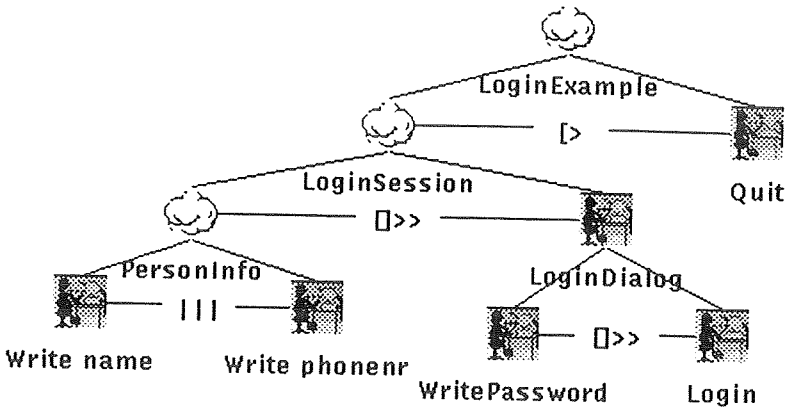


Figure 8 A tree with preconditions

An example of getting the preconditions

In this example we will use the task tree in Figure 9. We will search the task tree top-down, left to right, *for tasks that have an* ***enabling left brother***.

The first task we will find in the task tree with an enabling left brother is *LoginDialog*. The task *LoginDialog* has *PersonInfo* as a left enabling brother. *PersonInfo* is thus the precondition to *LoginDialog*. The results this far will thus be: *PersonInfo* is the precondition to *LoginDialog*.

Because *PersonInfo* is an abstract task we must check its children to find its preconditions. The method will find the task *Write name* with the interleaving operator and collect this to the result. The next child is *Write phonenr* (also interleaving) and we will also collect this task to the results. The results will be: *Write name* and *Write phonenr* are preconditions to the task *PersonInfo*.

The next task with a left enabling brother is *Login*. It has *WritePassword* as a left enabling brother. We will therefore write the task *WritePassword* as a precondition to the task *Login*.

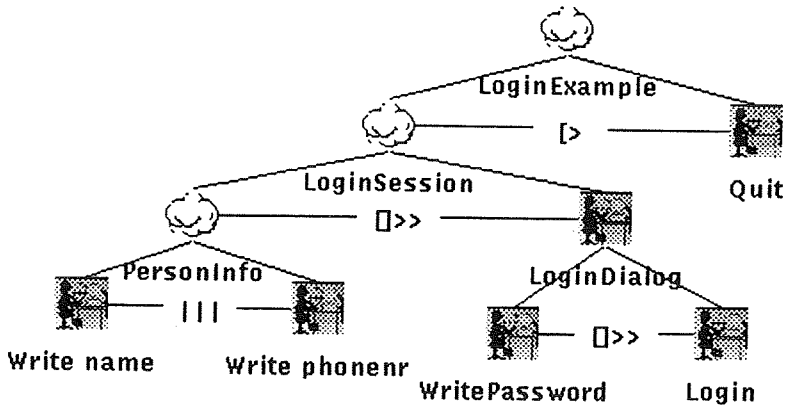


Figure 9 A tree with preconditions

The results of the above tree include thus the following:

18

- *WritePassword* is the precondition to *Login*
- *PersonInfo* is the precondition to *LoginDialog*
- *Write name* and *Write phonenr* are the preconditions to *PersonInfo*

In the precond-table we will write the following:

```
Login|WritePassword
LoginDialog|PersonInfo
PersonInfo|Write name|Write phonenr
```

The available tasks

When the user has succeeded to fulfil a task we get the available tasks from the task model. The available tasks are tasks the user can perform right after s/he has performed the current task. These tasks are not only interesting to the evaluator but also for supporting task-oriented help since the information gives the tasks the user can perform at the current state. This helps the user to see what is possible and what is not possible to do in the current state.

The evaluator can use the available tasks to understand the user's preferences when performing a task, and examine the application to see how well it supports the user's needs.

When we are searching the task tree for the available tasks we are only interested in the *interaction* tasks, because it is only these that the user can perform by interacting with the system. The method for getting the available tasks is used together with the method for deciding if the task's preconditions are fulfilled or not.

If the task's preconditions were not accomplished we will not count this task as an available task. This means that we can only get the correct available tasks during the evaluation process. It is only then we can know if a task was performed or not.

The evaluation method

The evaluation process starts with a log from the logfile. The following steps describe the method:

1) Get a log from the logfile

2) Does the log exist in the log-task table?

   Yes: Look up the associated task No: Error

3) Has the task preconditions?

   Yes: See if they are satisfied No: Task accomplished

4) Are the preconditions satisfied?

   Yes: Task accomplished No: Precondition Error

5) Is the accomplished task a precondition to another task?

Yes: Change the precondition table

19

6) Get the available tasks from the current state

An example of the evaluation method in use

The following example of the evaluation method uses the task model shown in Figure 10 below. The task model describes a user interface where it is possible to perform a reservation of a movie by inserting a name and choosing a movie. The reservation is then executed with the task *Book*. There is also a search possibility, to search for movies.

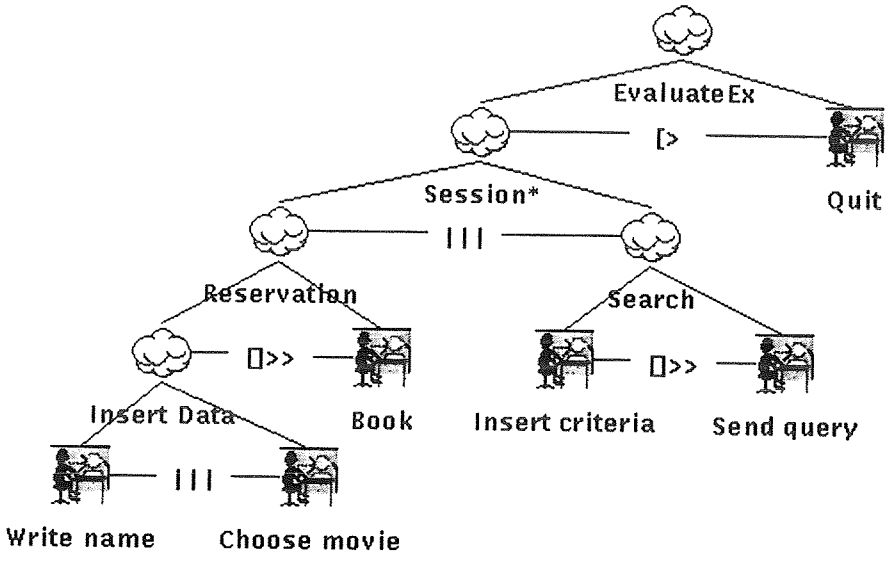


Figure 10 The task tree of the example

The first thing we do in the evaluation process is to read a log from the log file (in this example only the important part of the logs are shown, coordinates e.g. are omitted):

```
1. click {\Canvas 1}
```

This log is not found in the log-task table, i.e. it is not associated with any task and we will record this as an error.

This error occurred because the designer did not specify this log in the log-task table. In this case the user clicked on an image thinking that something would happen. These types of errors occur when the user clicks on or uses items that exist in the user interface but lack function, e.g. things like images, labels or text strings.

```
2. click {\Send query}
```

This log is associated in the log-task table with the task *Send query*. First we check if the task has preconditions by checking the precond-table:

```
Send query|Insert criteria
```

The answer is yes, the task has a precondition. Since we do not find *true* after *Send query* the precondition of the task is not satisfied. This user action we will log as a **precondition error**, because the precondition of the task the user tried to perform, was not satisfied. That is, the user tried to perform the task *Send query* before he had inserted the search criteria.

```
3. click {/Insert criteria}
```


20

The next log is `click {/Insert criteria}` which is associated with the task *Insert criteria* (to insert a search criteria) in the log-task-table. This task does not have any preconditions so the task is completed or with other words achieved. Because the task was accomplished we now get the available tasks.

The available tasks from this state are *Send query, Write name, Choose movie* and *Quit.*

However, the task *Insert criteria* is itself a precondition to another task, *Send query.* This means that we must change the precond-table, the field `Insert criteria` to `true`:

```
Send query|true
```

```
4. click {\Send query}
```

The next log is the same as above. When we check the precond-table this time, because of the previous log, we will find that the precondition for this task now is satisfied. The user has now successfully performed the task *Send query* and we will now return the available tasks from the current state.

```
5. text {\Namefield}
```

Log nr 5 is found in the log-task-table and the associated task is *Write name.* The task does not have a precondition and is therefore accomplished. The task is though a precondition *itself.* It is precondition to the abstract task *Insert data.*

The available tasks from this state are *Book, Write name, Insert criteria, Quit* and *Choose movie.*

Since the task *Insert data* is abstract we must continue. An abstract task can not be performed by interaction from the user so we (the method) must change its fields in the precond-table. That is, in the precond-table we change the field of the precondition for *Insert data* to `true`.

```
Insert data|true|Choose movie
```

```
6. click {\Independence day}
```

The next log is associated with the task *Choose movie.* As in the previous step the task does not have a precondition and it is therefore accomplished, but it is a precondition itself. We will change the field of the precondition in the precond-table:

```
Insert data|true|true
```

```
7. click {\Book}
```

The next log is associated with the task *Book.* It has the precondition *Insert data* which is, due to the previous step, satisfied (all fields are true). Thus we will record *Book* as a accomplished task with its precondition *Insert data* satisfied.

The results from the evaluation

The results from our evaluation method concern how well users succeed to perform their tasks, and the errors the users perform. The results, derived from the evaluation method and from the logs provided by Replay, include the following:

- The accomplished and failed tasks, and those never tried

21

- The user errors of two kinds
- Numerical and temporal information of the above. For example, we can get how many times the tasks and the errors were performed and the temporal order of the tasks and the errors.
- How long time each task took to complete. This is calculated based on the time stamp that exist on every log, telling how long time proceeded since the last log was performed, information provided by the recording tool (Replay).

  The times for the abstract tasks (not performed by the user) are calculated by summing up the time of the abstract task's children. The information about the completion time is useful when measuring efficiency.
- When (at what time) in the interaction, the errors occurred. If the errors only ocurred in the beginning it could mean that the user should have more support and help in the beginning of the interaction.
- Task patterns, i.e. sequences of accomplished tasks that occurs more than one time. This information can be used to identify where macros should be implemented or where some automations should be done.

  For example, if the task "Choose printer" always is followed by the task "Print a file", the task "Choose printer" could be done or set automatically by the program.
- The available tasks from the current state, i.e. from a certain point in the logs.
- Other useful information, such as the test time, the time when the errors occurred, how many times the scrollbar was used and how many times the window was resized.

These results are then used in the USINE tool (see part 5) where we present them in different ways.

The accomplished, failed and never tried tasks

The tasks the user accomplished, failed and never tried during the user test are counted and recorded. The accomplished tasks are of two kinds, those with preconditions and those without.

As failed tasks we count those tasks the user tried to perform but failed because the preconditions were not satisfied. In this case the user tried to perform a task but the task's preconditions were not satisfied.

If the user fails to perform a task (i.e. makes an error) it is difficult to know which task s/he intended to perform. A way to get to know which tasks the user intends to perform is to ask the user before the user test "what task do you want to perform now?". Another possibility exists when the task has preconditions. If the user performs an error we can then see which tasks the user wanted to perform and why s/he failed.

Consider the following example: the task *Choose a file* is a precondition to the task *Print*, and the task *Print* is executed by pushing the "Print-button". If the user in this case tries to perform the task *Print* without performing the task *Choose a file* first, we will know that the user's *intention* was to perform the task *Print*, but s/he failed because the precondition (the task *Choose a file*) was not satisfied.

The information about the tasks the user accomplished is useful when usability goals have been specified (as mentioned in part 2). For example, if the goal is that the user must fulfil

22

certain tasks, it is possible to see if the goal was satisfied after the evaluation. It is possible to view the results in the tasks' temporal order, or ordered by task, in our tool USINE (see part 5).

The frequency of the tasks performed can tell the designer if the current layout of the user interface is optimal. For example, if some tasks in the same layout window are performed frequently their corresponding widgets should be close to each other or grouped in the layout. This would let the user move the mouse less, which would make the interface more efficient (Sears 1995).

Likewise, the corresponding widgets for the tasks that never or very seldom are performed, could either be hidden or be removed. For example, different options that are available when printing a file like setting the printer's name, which paper tray to use, etc. could be hidden in an option dialogue box.

The user errors

The user errors are of two kinds. The first kind consists of actions needed to perform a task but the preconditions were not fulfilled (precond-errors). The second kind consists of actions not needed to perform any task. These actions do not exist in the log-task-table because they are not associated with any tasks. The user thought these actions were useful to perform. For example, clicking on images that looked like buttons, clicking on labels or other things that exist in the user interface but lack a function.

This information gives a possibility to conclude how *easy* the user can perform her/his tasks, and which tasks caused problems for the user. Errors are often a part of the usability goals (Dumas & Redish, 1994), e.g., a goal could be that the user must only do 5 errors.

Goals like this could be useful when you want to compare two alternative designs. The alternative to be chosen should thus have the least number of errors from the evaluation. Thus the information from the evaluation method provides the necessary information to measure if such goals have been achieved.

Other information

The time when the user errors occurred is also recorded to see when during the user test the most errors occurred. Other useful information which could indicate readability problems with the interface is also included in the results. This includes how many times the user used the scrollbar and how many times the window was resized and moved. Finally, the time the user test lasted is also recorded.

5. The tools to support the method

To support the method described in part 4 we use two tools, QC/Replay and USINE. QC/Replay is a commercial software while USINE is the tool developed as a part of this work.

To record the logs from the user test we use QC/Replay (Replay) version 2.5 from CenterLine Software[1]. Replay saves the user's actions in test scripts, which capture the

[1] Internet: http://www.centerline.com/products/qc_rply.html

23

user's behaviour as s/he affects the widgets (buttons, menus, lists, etc.) of the user interface. The scripts can then be replayed to examine the user's behaviour during the test. Replay also includes other features as naming widgets, taking snapshots of the screen and running test scripts using Tcl (an interpreted command language).

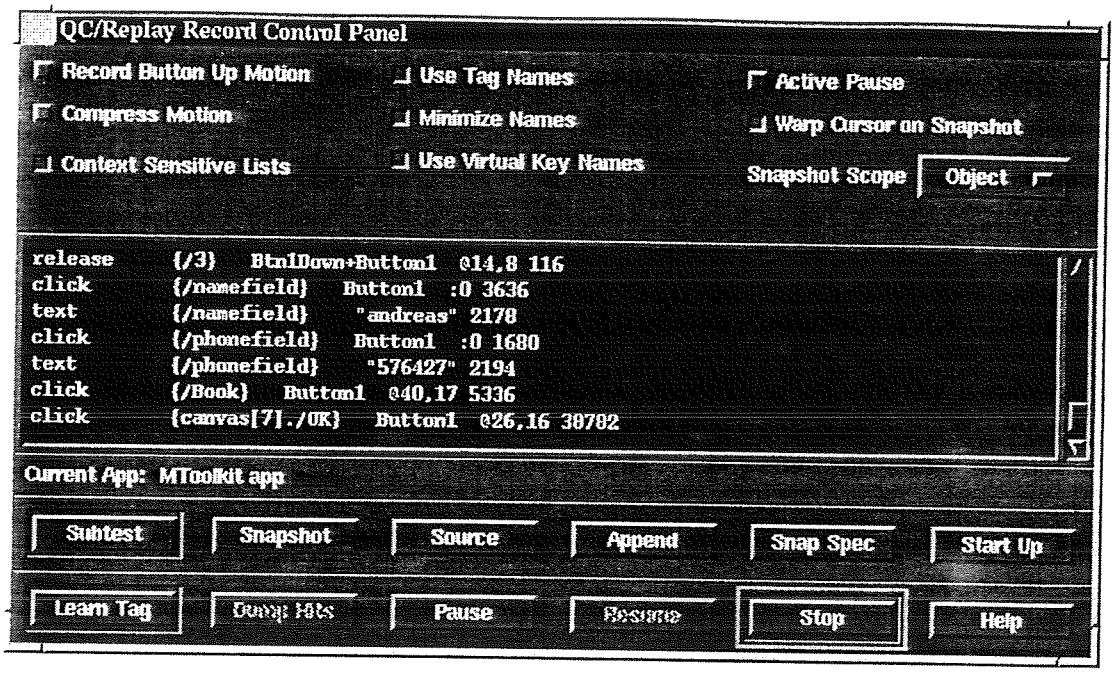


Figure 11 The QC/Replay Record Control Panel

The recordings given by the tool (see Figure 11 above) include pressing and releasing the mouse, double-clicks and keys pressed. A typical log (one line) includes:

- The type of the user action such as a mouse click
- The name of the widget affected, the target of the action, e.g., a list
- The mouse button used
- The content of the widget affected, such as an item of a list
- The coordinates of the mouse click
- The time elapsed from the previous log

The advantage of Replay is that it saves the recordings in a readable and editable file. This makes it possible to use the logs as input for our tool. Another advantage is the possibility to name the different widgets of the application as wished.

The tool (USINE - **US**er **IN**terface **E**valuator) we have developed automates and facilitates the evaluation process. It is implemented in Java which makes it platform independent. The tool is divided into two parts, the preparation part and the result part. The *preparation part* is where the designer creates the log-task table and provides the names of the different files needed for the evaluation. These files are:

1. The log file
2. The file with the task model

24

3. The file with the log-task-table

The *result part* contains the results from the evaluation and possibilities for the user to present it in different ways.

The preparation part

The preparation part supports the creation of the log-task table as described in part 4. The designer can load the tasks from the task model of the application on the right side and the logfile from the user test on the left side (see Figure 12). The designer then associates each task with a log that executes it. The tasks are indented depending on their level in the task tree, the more right the deeper level. It is also possible to load an existing log-task-table to edit it.

Abstract, user and application tasks, that do not have any corresponding logs, are marked in the log-task-table with a "#". Optional tasks (see part 3), are marked within brackets "[]".

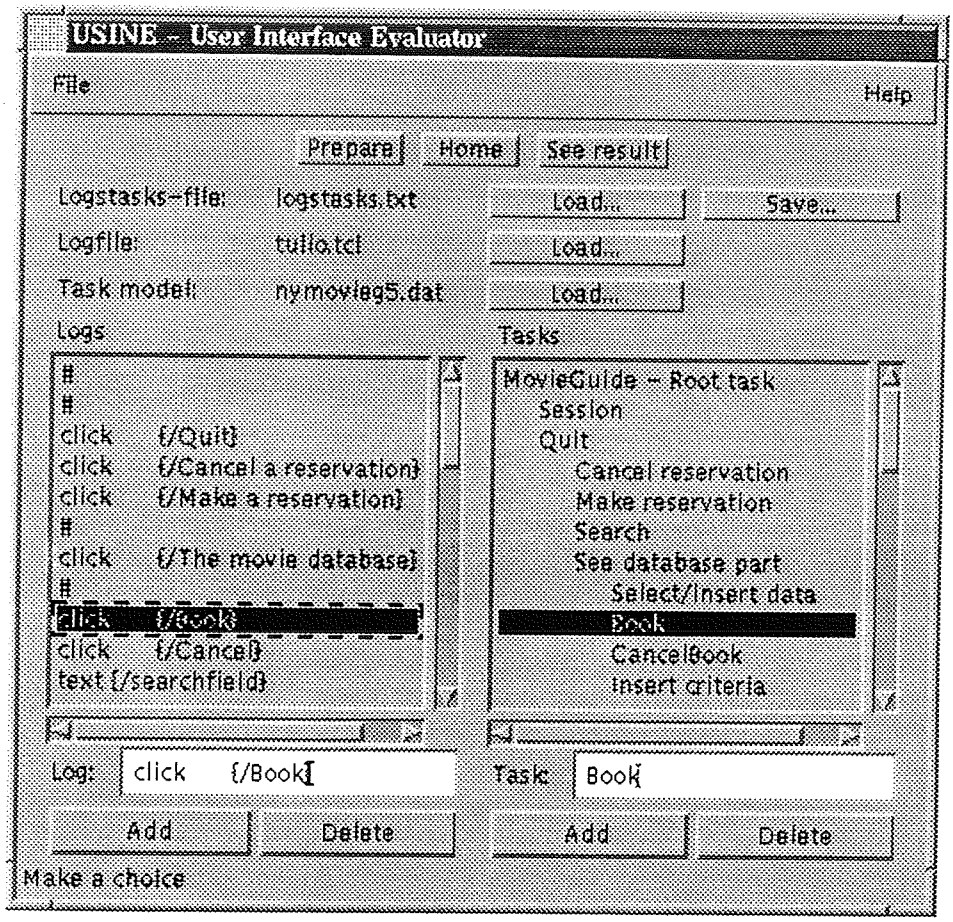


Figure 12 The preparation part of USINE

The result part

In the result part the user starts the evaluation by pressing the *Evaluate* button. The evaluation method of the tool is the one described in part 4. The evaluation is shown in the text area as it proceeds (see Figure 13). The results shown are the user logs in order and the type of the task or the error associated, depending on if the user succeeded to fulfil the task or not. If the user succeeded to fulfil a task, the following available tasks, possible to perform in the next step, are shown.

25

The different formats used in USINE (see Figure 13), when printing the results from the evaluation in temporal order, are the following:

- A task accomplished without preconditions, i.e. the task has no preconditions and is thus available at any time, e.g.:

```
Currentlog: click        {/The movie database}
Task "See database part" without preconditions achieved
Available tasks:
Choose viewstyle, Choose director, Choose category, Premiers, All
movies, Choose a movie, Done
```

- A task accomplished with preconditions, i.e. the task is accomplished with its preconditions satisfied, e.g.:

```
Currentlog: click        {/timelist}
Task "Choose time" with precondition "Choose cinema" achieved
Available tasks:
Choose time, TheMovie, Choose the movie, [Where to sit],
Select/insert data, SelData, Choose cinema, Write a name, Write
phonenr, [Nr of persons], CancelBook
```

- A precondition error, i.e. the user tried to fulfil a task but its preconditions were not satisfied, e.g.:

```
Currentlog: click        {/timelist}
-----> Preconderror: the precond "Choose cinema" not fulfilled for
the task: "Choose time"
```

- Other errors, i.e. not precondition errors and not associated with any task, for example clicking on images or other things that lack a function in the interface, e.g.:

```
-----> Error (type1): click        {canvas[2].canvas}
```


26

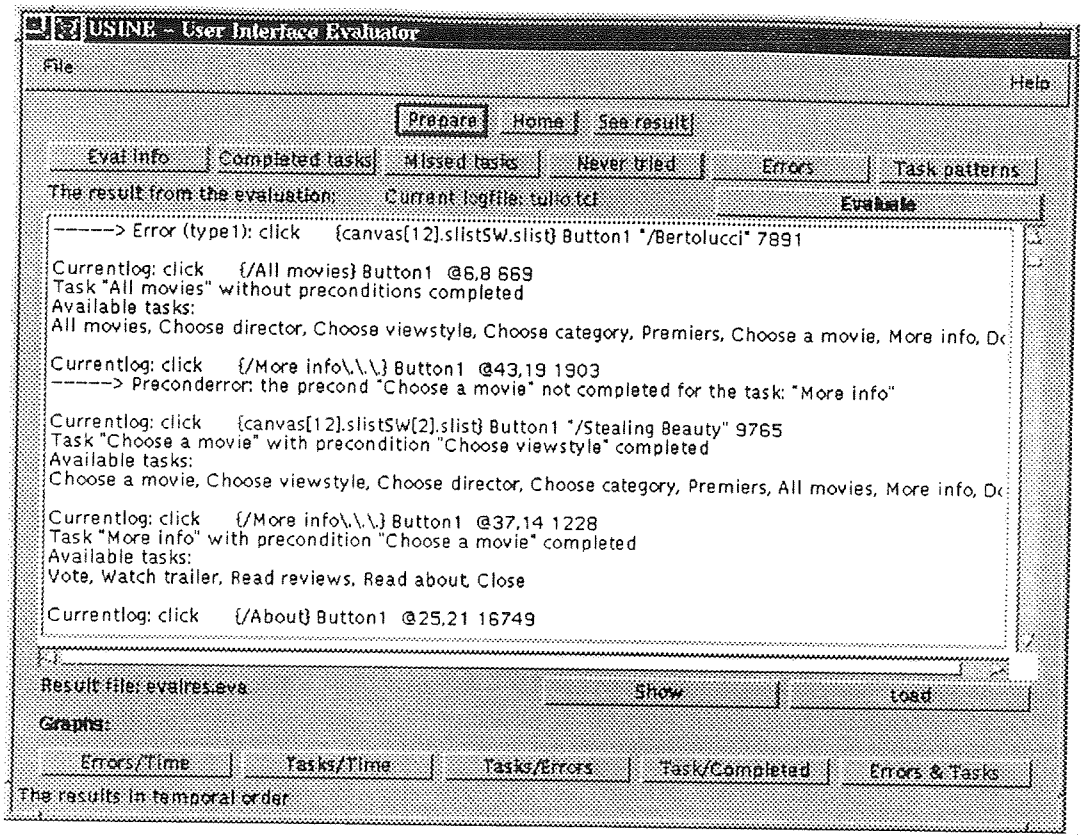


Figure 13 The result part of the evaluation

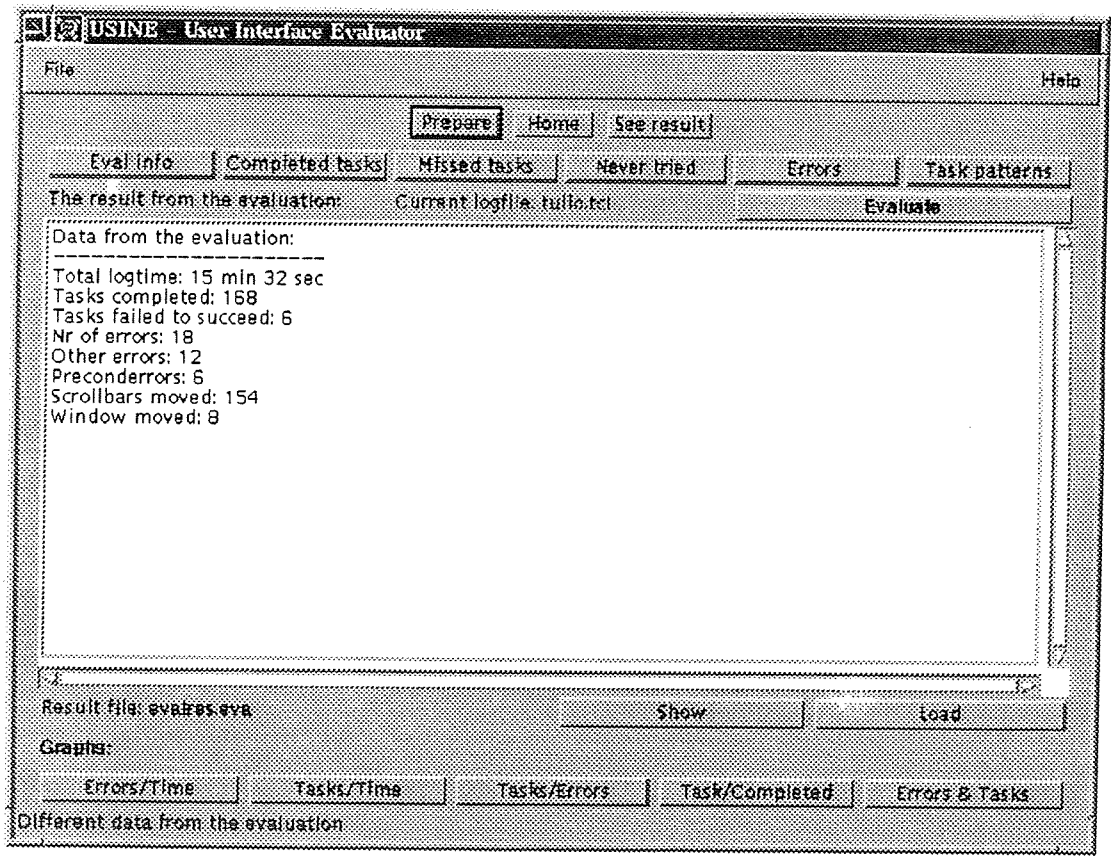


Figure 14 The data from an evaluation

27

The upper button set provides different ways for the user to present the results:

- The *Eval info* button gives various data from the evaluation such as the number of the accomplished and missed tasks, the number of errors and the time the test lasted (see Figure 14).
- The *Accomplished tasks* button gives the accomplished tasks and how many times they are performed. The frequency of the tasks are useful when deciding the layout of the interface. To make the interface more efficient tasks performed frequently (in the same window of the layout) should be close to each other (Sears 1995).
- The *Missed tasks* button gives the tasks the user tried to perform but failed beacuse their preconditions were not satisfied, and how many times each task failed.
- The *Never tried* button gives the tasks the user never tried to perform.
- The *Errors* button gives all the errors divided into precondition errors and other.
- The *Task patterns* button gives the found task patterns among the accomplished tasks (see Figure 15). That is, found sequences of tasks among the accomplished tasks.

The *Show* button (below the result window) gives the entire result from the evaluation in temporal order. It is also possible to save this result in a file and load (with the *Load* button) at a later moment.

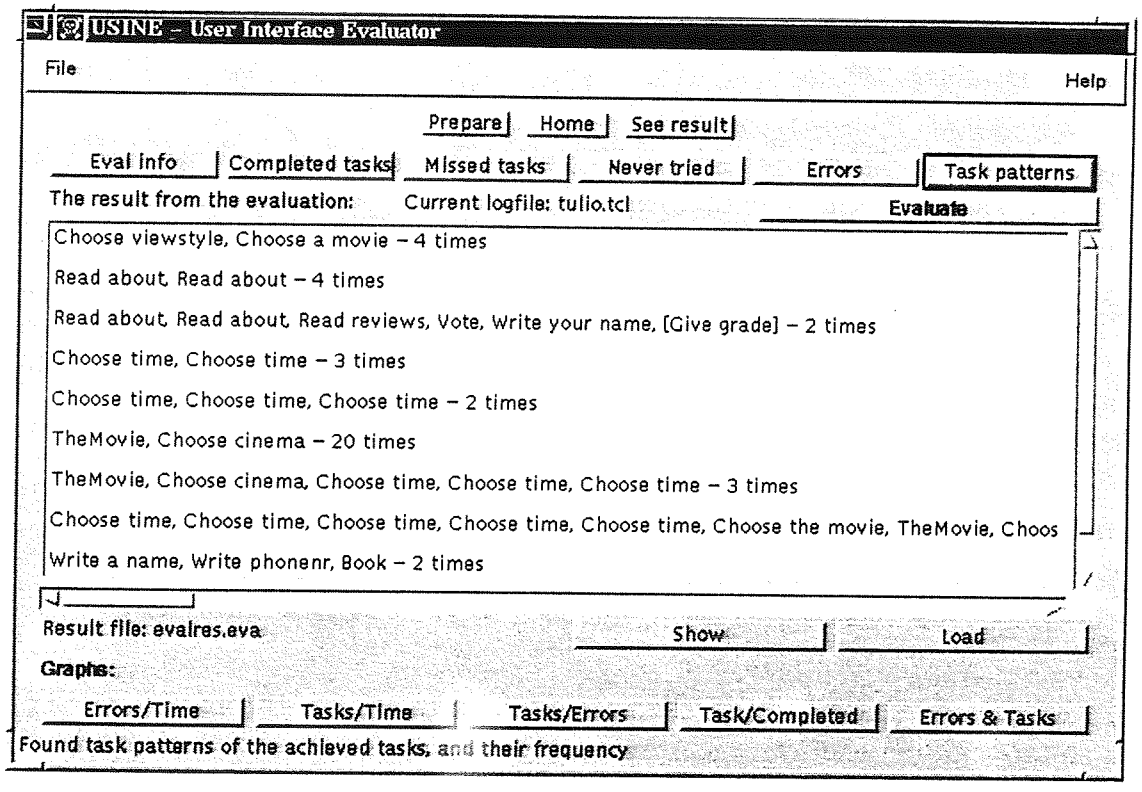


Figure 15 Task patterns, the pattern and how many times they occurred

Finally, the lower button set, provides different graphs showing the data from the evaluation in different manners:

- The *Errors/Time* button shows a graph with the number of errors on the y-scale and the time on the x-scale (see Figure 16).

28

- The *Tasks/Time* button shows a chart graph with the tasks on the x-scale and how long time they took to perform on the y-scale (see Figure 17).
- The *Task/Errors* button shows a chart graph containing the number of times the tasks were performed.
- The *Tasks/Accomplished* button shows a chart graph containing the number of precondition errors associated to each task.
- The *Errors & Tasks* button shows one pie chart containing the different types of errors and their percentage, and one pie chart containing the number of the tasks accomplished, the tasks missed and those never attempted (see Figure 18).

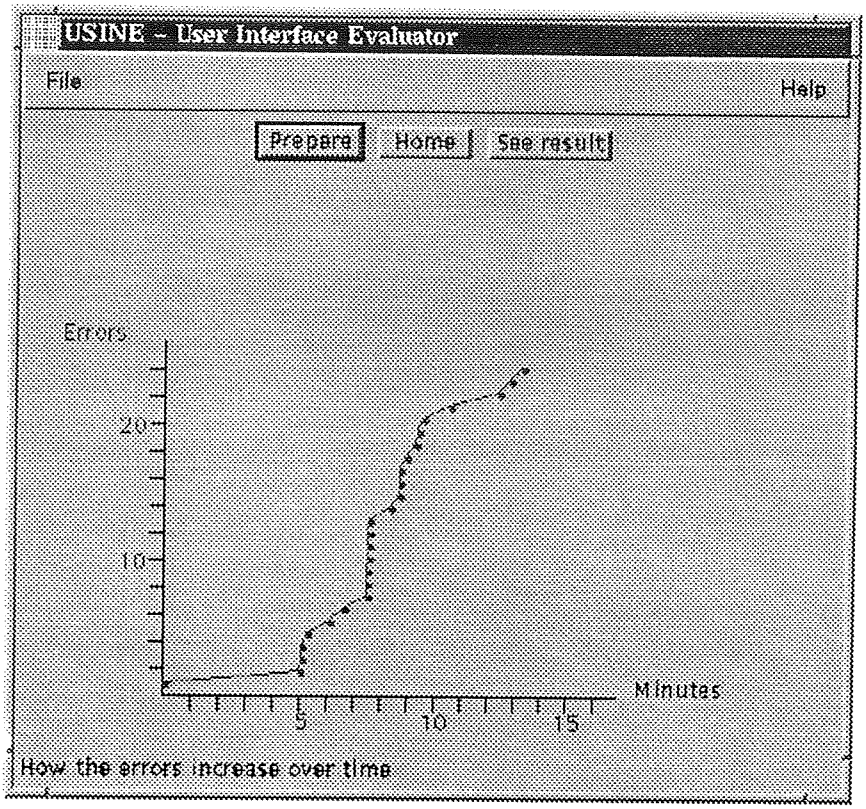


Figure 16 When the errors occurred during the user test

29

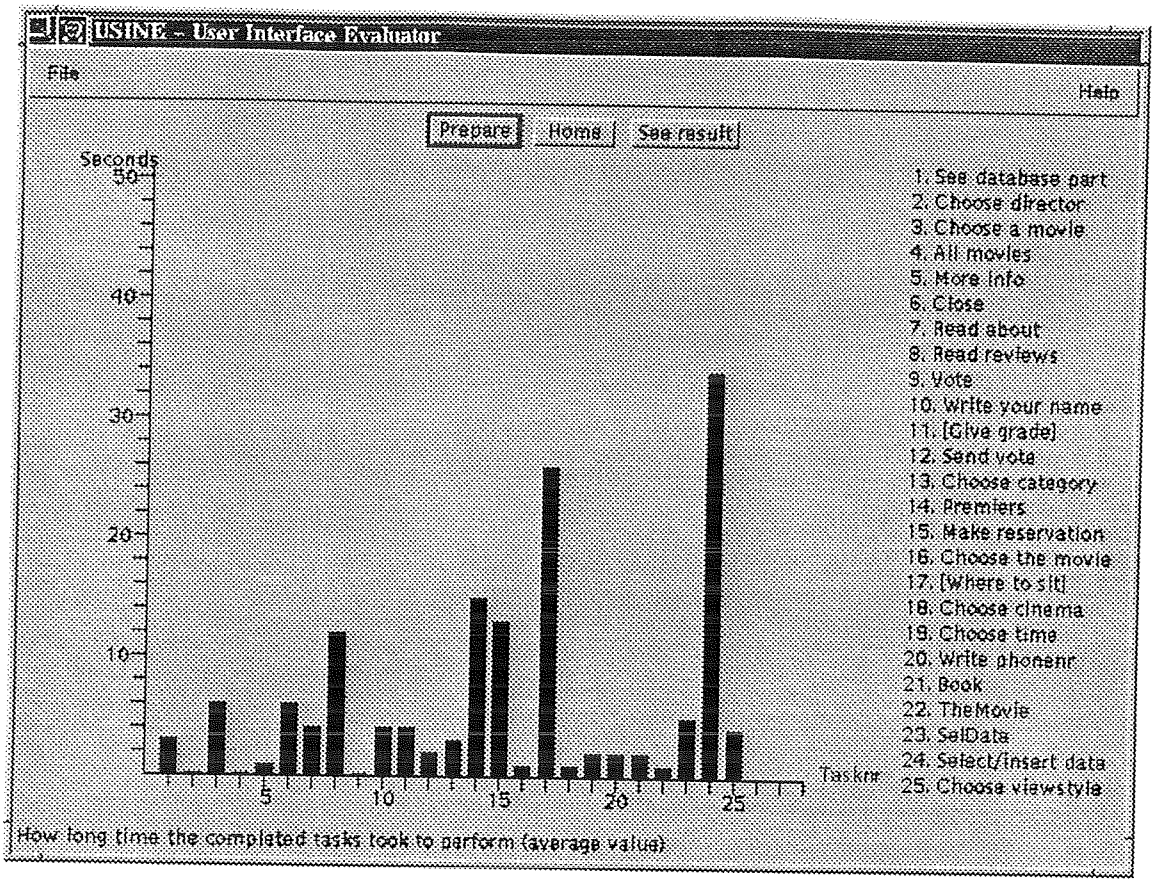


Figure 17 How long time each task took to perform

30

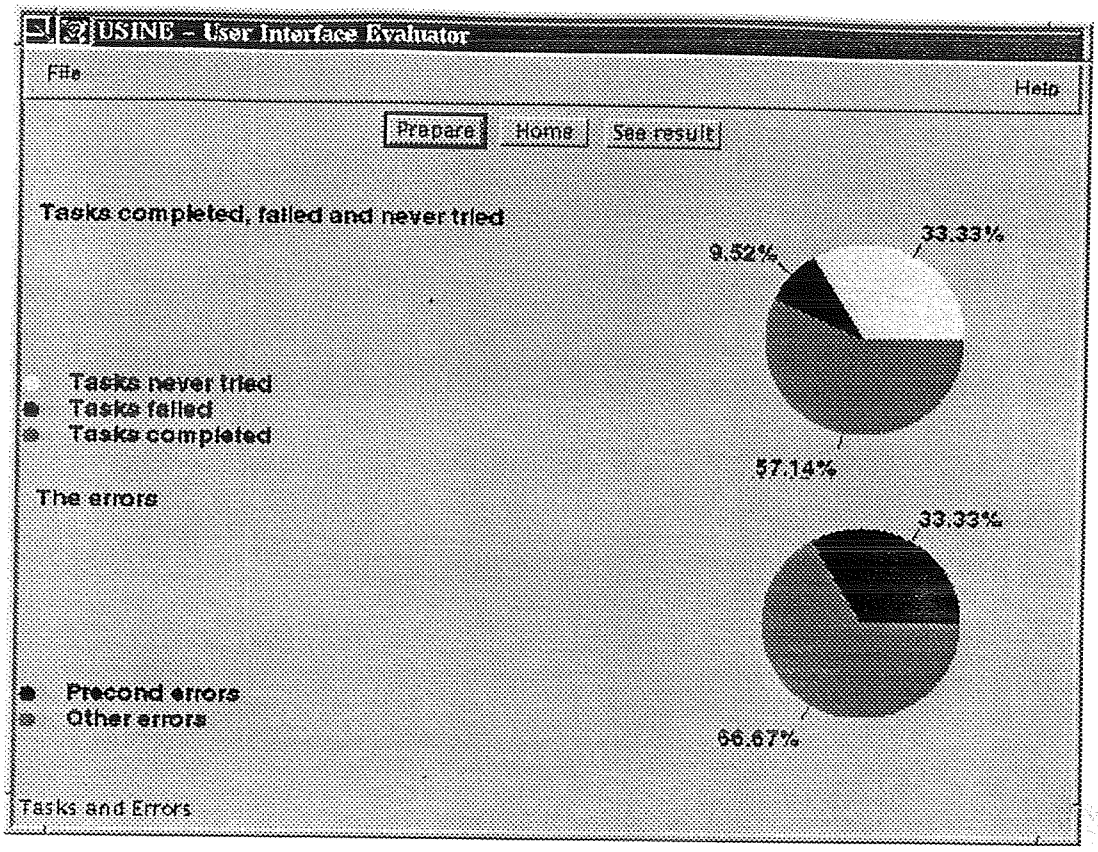


Figure 18 Pie charts of the tasks and the errors

6. An example of application

To apply the proposed evaluation method we chose to evaluate the application *MovieGuide* which is a Java application. In this example we used it in the X Window System environment (UNIX).

The user interface of the application

The application MovieGuide (see Figure 19) consists of one part with a database of movies, where it is also possible to get information about movies, and one part where it is possible to make reservations for movie tickets. Furthermore it is also possible to cancel a reservation and to search for movies in the database.

31

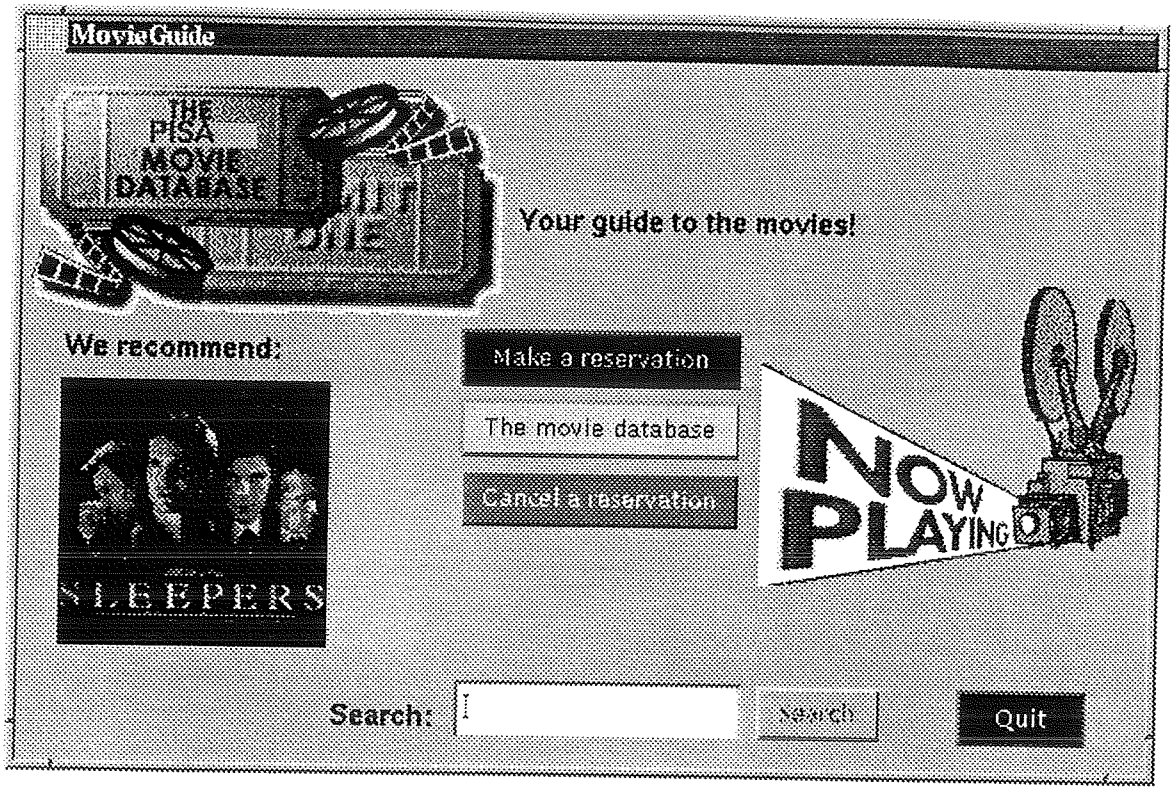


Figure 19 The MovieGuide, main window

The database part gives the opportunity to search a movie by director, by first openings, by category or all at the same time.

When a movie is chosen it is possible to see a film clip from the movie, to read about the movie (the story, the actors, etc.), to give a vote for the movie and to see what other persons have voted.

The reservation part (see Figure 20) provides a possibility to choose a movie from a list and then make a reservation on a certain cinema, on a certain time and for a number of persons. It is also possible to decide where in the cinema you want to sit, in the front, in the middle or in the back.

32

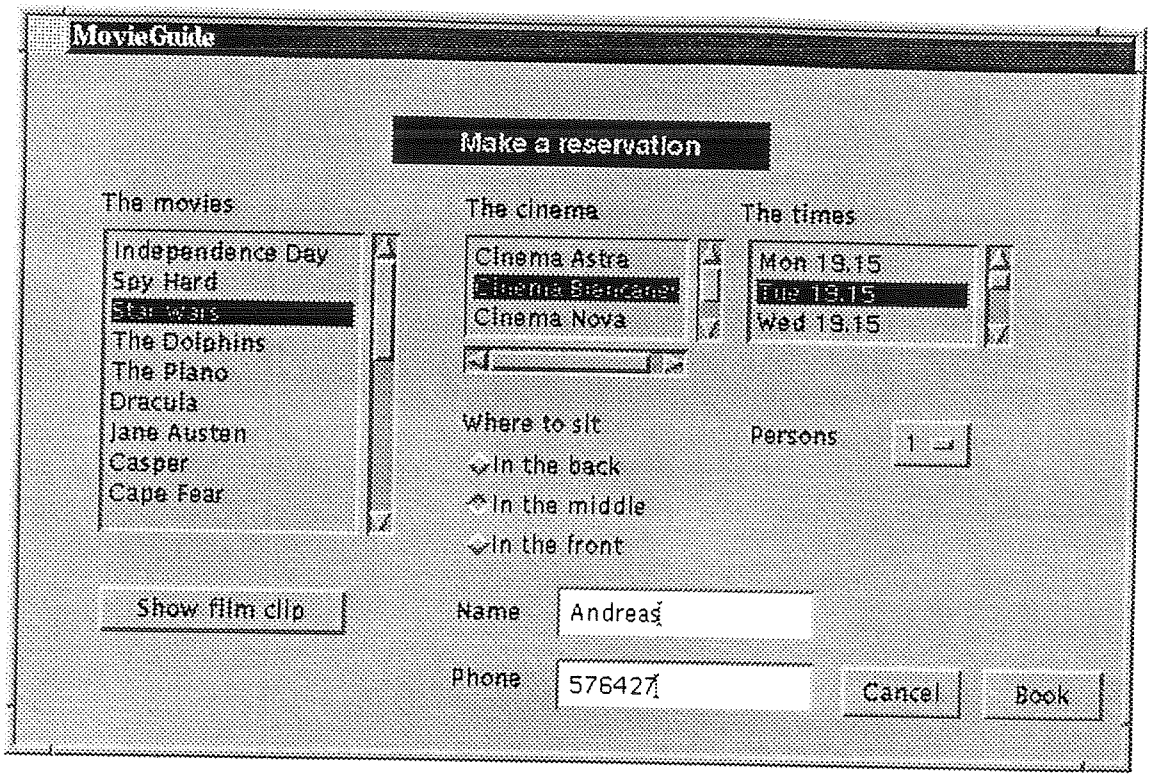


Figure 20 The reservation part of MovieGuide

The task model of the application

The task model of the application (see Figure 21) was made in the TLIM-editor using the task model described in part 3. The task model was made *according to* the existing user interface of the MovieGuide application. It was *not* made as a general task model to describe the different tasks that *could* be possible to perform. The reason for this is because we want to evaluate an existing interface not to develop a new one. The tasktree was made so deep that every interaction task has a corresponding log in the log-task table.

In the task tree we have interaction tasks with children. This is used to express that a new interaction part of the user interface is shown when the task is performed, e.g. a new window where the interaction can continue. For example, when the user has performed the task *Vote*, a new window will be opened where the interaction can continue. In this window the user can perform the tasks that are children to the *Vote* task.

Other characteristics of the task model include the following:

- The *Quit* task disables the whole session of the application.
- The task *Session* is divided into four parts with the choice operator between the parts. The different parts are: *Cancel reservation*, *Make reservation*, *Search* and *See database part*.

  The choice operator makes that when one of the children of *Session* is performed their brothers are not available any more.
- The *Cancel reservation* and the *Send query* task are not derived completely in the task model because they were not evaluated in this example. That is, it is not necessary to

33

include parts of the user interface, not being evaluated, in the task model. Thus the designer only has to specify the parts that she wants to evaluate.

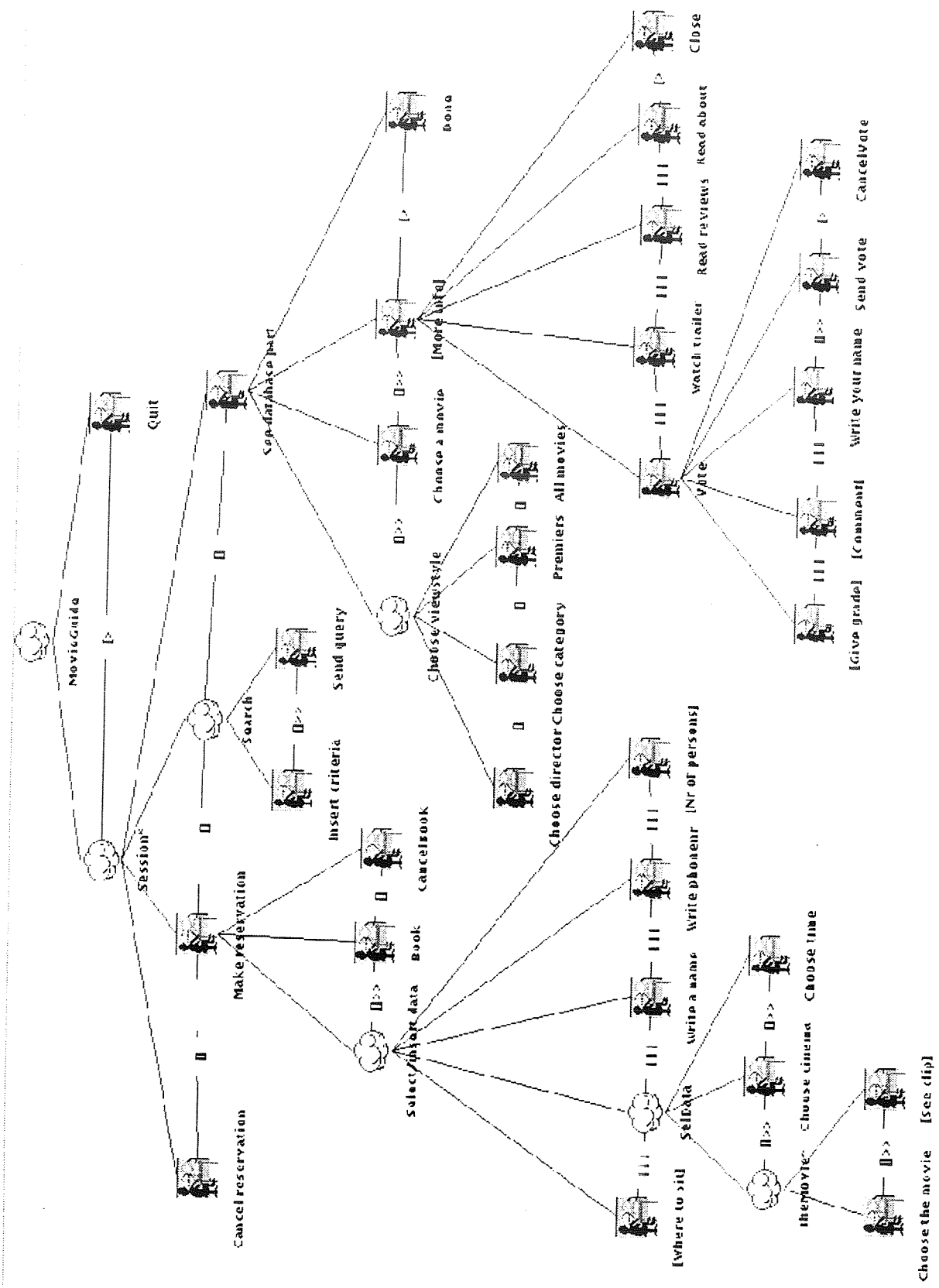
- To be able to perform the task *Book* the user has to fill in necessary data before. These data include the movie, the cinema, the time the movie is shown, the name and the telephone-number. *CancelBook* disables the possibility to perform the reservation (the booking).

- To choose *More info* the user must first perform the task *Choose a movie*, but before this is possible the task *Choose viewstyle* must be performed, i.e. how the available movies should be presented to the user. After the viewstyle is chosen it is possible to perform the task *Choose a movie*, enabling the task *More info*.

- Some tasks are optional, e.g. *Where to sit* and *Nr of persons*. This means the user does not have to perform these tasks. For example, the task *Nr of persons*, has the default value "1". If the user is satisfied with "1" she will not do anything, i.e. no interaction with the interface will occur.

Figure 21 The task model of MovieGuide(next page)

34


MovieGuide
Quit
Session
Cancel reservation
Make reservation
Search
Insert criteria
Send query
Book
Cancelbook
Select right data
See/Change part
Choose viewstyle
Choose a movie
[More info]
Choose director
Choose category
Premiers
All movies
Vote
Watch trailer
Read reviews
Read about
Close
[Give grade]
[Comment]
Write your name
Send vote
CancelVote
[where to sit]
SelData
Write a name
Write phonenr
[Nr of persons]
Choose cinema
Choose time
Choose the movie
[See clip]

35

The enabling tasks of the task tree, or the precondition tasks, are the following (according to the algorithm in part 4):

- Select/insert data *enables* Book
- SelData, Write a name and Write phonenr *enables* Select/insert data
- Choose time *enables* SelData
- Choose cinema *enables* Choose time
- TheMovie *enables* Choose cinema
- Choose the Movie *enables* TheMovie
- Choose the movie *enables* [See clip]
- Insert criteria *enables* Send query
- Choose a movie *enables* More info
- Choose viewstyle *enables* Choose a movie
- *One of the tasks*, Choose director, Choose category, Premiers or All movies, *enables* Choose viewstyle
- Write your name *enables* Send vote

The disabling tasks are:

- Quit *disables* Session
- CancelBook *disables* Book
- Done *disables* More info
- Close *disables* Read about
- CancelVote *disables* Send vote

The precondition table of the example

From the tasktree we automatically create the precondition table by the method described in part 4. This gives the following precondition table of the task model (read as, `x is a precondition to | y`):

```
Select/insert data|Book
Choose time|SelData
Write a name|Write phonenr|SelData|Select/insert data
Insert criteria|Send query
Choose viewstyle|Choose a movie
Choose director|Choose viewstyle
Choose category|Choose viewstyle
Premiers|Choose viewstyle
All movies|Choose viewstyle
Choose a movie|More info
TheMovie|Choose cinema
Choose the movie|TheMovie
Choose cinema|Choose time
Write your name|Send vote
Choose the movie|[See clip]
```


36

In the precondition table we can see that *Select/insert data* has three preconditions, *Write a name, Write phonenr* and *SelData*. The task *Choose viewstyle* on the other hand, needs only one of the tasks, *Choose director, Choose category, Premiers,* or *All movies* to be accomplished.

The log-task-table of the example

The log-task-table was created by with each task, associate a log from the logfile. The abstract tasks were marked with the "#". This gives the following log-task-table:

```
#|MovieGuide
#|Session
click       {/Quit}|Quit
click       {/Cancel a reservation}|Cancel reservation
click       {/Make a reservation}|Make reservation
#|Search
click       {/The movie database}|See database part
#|Select/insert data
click       {/Book}|Book
click       {/Cancel}|CancelBook
text    {/searchfield}|Insert criteria
click   {/Send}|Send query
#|Choose viewstyle
click       {canvas[12].slistSW[1].slist}|Choose a movie
click       {/More info\.\.\.}|More info
click     {/Done}|Done
click       {/In the front}|Where to sit]
#|SelData
text        {/namefield}|Write a name
text        {/phonefield}|Write phonenr
release       {/4}|Nr of persons]
click       {/Directors}|Choose director
click       {/Categories}|Choose category
click       {/This week premiers}|Premiers
click       {/All movies}|All movies
click       {/Vote}|Vote
click       {/Trailer}|Watch trailer
click       {/Reviews}|Read reviews
click       {/About}|Read about
click       {/Close}|Close
#|TheMovie
click       {/cinemalist}|Choose cinema
click       {/timelist}|Choose time
click       {/1}|[Give grade]
text        {textA}|[Comment]
text        {textfield}|Write your name
click       {/OK}|Send vote!
click       {/movielist}|Choose the movie
click       {/See film clip}|[See clip]
```

Figure 22 The log-task-table of the example

The user test of the application

The user test of the application was performed by six different users. The testers were computer science students in the age of 24 to 26 who had never seen the application before and therefore were total beginners. The users were given written instructions (see Figure 23) on a paper and an opportunity to ask any questions they liked before the test started. When the test had started questions were not answered directly, instead they were answered with a question. For example, if the user asked "What do I do now?" the answer would be of general type like "What do you think you could do"? No further help was given during the test.

37

MovieGuide

MovieGuide is a tool where you can get information about movies and make ticket reservation for the local cinemas.

Please perform the three following tasks:

1. You want to see a Bertolucci film but you have forgotten what the title was in English. Find the title.

2. Compare the two movies *Independence Day* and *Sleepers* by using the information the MovieGuide can give you.
You can:
- look at trailers,
- read about the movie and
- see reviews of the movie

Give your vote to the MovieGuide database.
One opinion for *Independence Day* and one for *Sleepers.*
- Which category do these two movies belong to?
- Are these movies premiers?

3. You and your friends have decided to go and watch either *Independence Day* or *Sleepers.* The only time you and your friends can go is Saturday 19.30.

Make a reservation for the movie that is shown at that time. Make the reservation for you and your three friends. You want to sit in the front.

Figure 23 The instruction for the users

The test consisted of three tasks the user had to perform. The tasks were chosen to force the user to examine different features of the application. During the user test the logs were recorded using the Replay tool (see part 5). After the test the users answered to three questions concerning their impression of the application.

The evaluation of the application

The purpose of the evaluation was to improve the user interface. The aim was also to see which tasks the user performed and the errors s/he did. We did not specify any particular usability goals before we started the evaluation because we were not testing *towards* a quantitative target, e.g. to see if the application was good enough.

The evaluation of the application can be divided into two different approaches. The first approach was based on observations made during the performance of the six user tests. An advantage of performing a user test is that you actually see the user using the application which can give you additional information about how to improve the user interface.

The second approach, using the *task model* and the proposed *method* in section 4, was performed after the user tests by using the USINE tool. We discuss below the benefits of the two approaches and their results. The first results from the evaluation however, consist of the users' answers to the questions, asked right after a test was finished.

38

The answers to the questions

To capture the users' thoughts and their subjective meaning we performed a smaller questionnaire. The questionnaire was not meant to be exhaustive but to give a smaller picture of the users' opinions. The questions asked to the participants after the user test were the following:

1. Did you have any problems during the test?
2. Which part was the most difficult (if any), or the most irritating?
3. Would you use this software (an improved version) in the future? Why, or why not?

A summary of the answers includes the following:

1. Did not get any feedback when moving the mouse.

   Nothing is shown in the lists when you go to the database part.

   Too small scrollbars in the cinemalist and the timelist.

2. The images, thought they were buttons.

   The timelist, was too small.

3. Yes, probably, it is useful

The answers to the questions cover some of the critical usability problems with the application; problems with the lists, the scrollbars, the images and the buttons. That is, the quality and the relevance of the answers were high, probably due to that the participants were computer science students. If the participants would not have had any knowledge of computer science it is likely that the answers would be different and of lower quality.

Evaluation based on observations

With "evaluation based on observations" we mean that we observed the users during the user tests, what they did and what they said, and what problems they seemed to have.

Based on this data, i.e., *observations* of the users during their use of the application, we tried to draw some conclusions how the user interface should be changed and improved. These *empirical* conclusions are based on the answers to the questions above and on our background knowledge of user interface evaluation.

The most important observations were the following:

- The users clicked on images because they thought they were buttons.
- The user clicked in empty lists because they did not understand how to get the information to the list. For example, to get the list of the times when the movies were shown you first had to choose a cinema.

  Another problem was that the lists (containing movies, etc.), when shown at the *first* time, were empty.

- Double-clicking on items, where a single click was sufficient.
- The scrollbars in the timelist were too small.

That is, an improved version of the application should include improvements on these issues. However, the results above can not say if the usability goals were satisfied or not, because we do not

39

have any measurable data. We "only" have some suggestions how the user interface could be improved.

Evaluation by the proposed method

Based on the task model and the user logs we performed an evaluation according to the method described in part 4. We used the USINE tool (see part 5) to get evaluation data from the tests. The average results were the following:

Accomplished tasks: 24

Failed tasks: 13

Never tried tasks: 5

Precond-errors: 13

Other errors: 7

Scrollbars moved: 56

Window resized: 9

Test time: 14 min 21 sec

We found that all users accomplished to fulfil all the tasks described in the instructions (see Figure 23). This indicates that the user interface was easy to use. The time it took the users to perform the tasks was too long though. This means that some sort of built-in help could be motivated.

The task that took the longest *time* to perform was the abstract task *Select/insert data*, i.e. to insert the different needed data for a reservation.

Among the *task patterns* we found that the user always chose a cinema after she had chosen a movie. This is an indication that the application automatically should provide the cinemas after the user has chosen a movie. We also got the information how many *times* each task was performed. It showed out that the tasks *Choose a movie* and *Choose cinema* were performed most.

The scrollbars were used frequently and the window was resized a few times. This could indicate a readability problem, i.e., the user had problems with reading and finding the wanted information.

The precondition errors

The most frequent precondition errors, i.e. the tasks causing the users problems were *Choose Time*, *More info, Send Vote* and *Book*. To understand better what the problems with these tasks were we can look at the results from the evaluation to see which precondition the user failed to perform before the current task.

The reasons for the precondition errors (failed tasks) can guide us how to improve the user interface, e.g. where to provide better information and more help.

This is possible because USINE gives us the result in the following format when a precond-error occurs, the task the user tried to achieve and the reason for the error.

```
Currentlog: click       {/More info\.\.\.}
-----> Preconderror: the precond "Choose a movie" not fulfilled for the
task: "More info"
```


40

This above means that the user did not, as required, perform the task *Choose a movie* before the current task, *More info*. When a task with a precondition is fulfilled we will get a result like below, including the available tasks:

```
Currentlog: click       {/More info\.\.\.}
Task "More info" with precondition "Choose a movie" achieved
Available tasks: Vote, Watch trailer, Read reviews, Read about, Close
```

The approach is thus to see which precondition (or preconditions) was not satisfied when the user failed to fulfil a task. The task (or tasks) associated with the unsatisfied precondition indicates then where improvements should be done in the user interface.

The reasons for the precondition errors follows below, together with the improvements that could be made to prevent the errors:

- For *Choose Time* it was because the user had to choose a cinema before it was possible to choose the time. The user obviously tried to choose the time before the cinema was chosen.

  An improvement of the interface could thus be to make it more obvious that a cinema must be chosen before a time. Or to show a cinema **and** a time after a movie has been chosen.

- The error of *More info* was due to that no movie was chosen before.

  To improve the interface a default value could be provided, e.g. a certain movie that always is chosen, together with more information and help how to perform the task *More info*.

  For example there could have been written that you must choose a movie before you can view more info.

- The errors of *Send Vote* were due to that no name had been inserted before sending the vote.

  Here is a lack of information. It should be more obvious that a name must be inserted before a vote can be given.

- The task *Book* failed because the user did not fill in the required fields like name, telephone-number, and so on.

  The improvements in this case could include message boxes ocurring each time the user tries to make a reservation. The messages should include exactly what field or fields are missing to perform the reservation. This task has many preconditions. To make it easier for the user to perform the required actions, a possibility could be to highlight the fields that must be completed.

The reasons for the errors do not say explicitly *how* to improve the interface though. However, they can tell that improvements must be done and indicate what must be done. The performed improvements must then be decided by the designer, guided and helped by these results.

41

The never tried tasks

Among the never tried tasks we found the task *See a film clip* located in the reservation part. We noticed that we had an inconsistency between the task *Trailer* in the "More info"-window and this task. These tasks are in fact the same.

This failure in design could be very confusing for the user, who may wonder what the difference is. We decided therefore to remove this task as it is not belonging to the other tasks in this part. That is, it is not a part of the tasks you normally perform when you want to make a reservation.

Other errors

The other errors were mainly caused by clicking on images. This was probably caused by, as mentioned above, lacking feedback when moving the mouse. With other words, the mouse pointer should be of one kind when moved over buttons, and of another kind when moved over a "non-clickable" area.

The temporal order of the tasks

By studying the temporal order the tasks were performed we saw that the order expected by the designer, was not always used. For example, the designer believed that the users would first choose a movie, then a cinema, and finally the time the movie was shown.

It showed out though, that the users could choose the cinema or the time as their first choice and therefore cause precond-errors. An improvement of the user interface could thus include a possibility for the user to choose what *s/he* wants first, or it could be more obvious what must be done first.

Comparison of the two evaluation approaches and their results

The two approaches found some similar usability problems with the product. However, the results from the evaluation based on the observations are *supported* and more specified by the results from the evaluation done with our method. For example, with our method we found certain tasks that had caused problems. This means we have something concrete to work with, not just a feeling that something is wrong with the current part of the interface.

Through the observations we found that the user clicked in empty lists to get some information into the lists. With our method though, we found the same problem but we also got which task caused the error and which task the user should have performed before to avoid the error. With other words, we also got the reason for the error, i.e. the actual precondition was not satisfied. That is, we know exactly where in the interaction the problem occurred. This means that our method could be of valuable support when performing a usability test.

Our method gives more specified information though, e.g. which tasks (and subtasks) were performed and how many times. This makes it possible to specify usability goals and to see if they were satisfied, something that is very hard to do from only observational data. The reason why we can set up usability goals is because we have *measurable* quantities of the user test.

Reasonable usability goals (as described in Dumas & Redish, 1994) of the application could e.g. be 2 precond-errors, 2 other errors, and all tasks achieved. With these goals in mind we can improve and re-design the application and perform the test again. If the goals are satisfied this time we have fulfilled the usability requirements of the application and can stop iterating in our design process.

The results from the proposed method can give indications how the designer should improve the user interface. We can also give more specific information, to say *where* in the interface improvements

42

should be done. This is due to the precondition errors, pointing out the tasks that caused the user problems.

However, the final decision, if improvements are to be made, and how they should be done, is still up to the designer.

The improved version of the application

Based on the evaluation above and the answers to the questions we decided to change the user interface as follows:

- To make the difference between buttons and images more distinct
- To make a default choice of a cinema and a time after the user has chosen a movie.
- To provide some feedback when moving the mouse over "clickable" respectively "non-clickable" objects
- To make the information visible in all the present list from the beginning
- To make the lists bigger
- To allow double-click on list items.
- To add more labels and messages, giving useful and helping information
- To take away the possibility to see a film clip in the reservation part

This information did not change the user interface drastically. That is, we did not perform major changes, like to change the structure of the program. Instead we made many smaller changes (as mentioned above) that users will notice when they actually use the application. The most visible changes were done in the reservation part, see Figure 24. The changes can be seen in the bigger lists and the removal of the "See film clip" button. Other changes include that some (default) information always is shown in the lists and a couple of instructive labels.

43

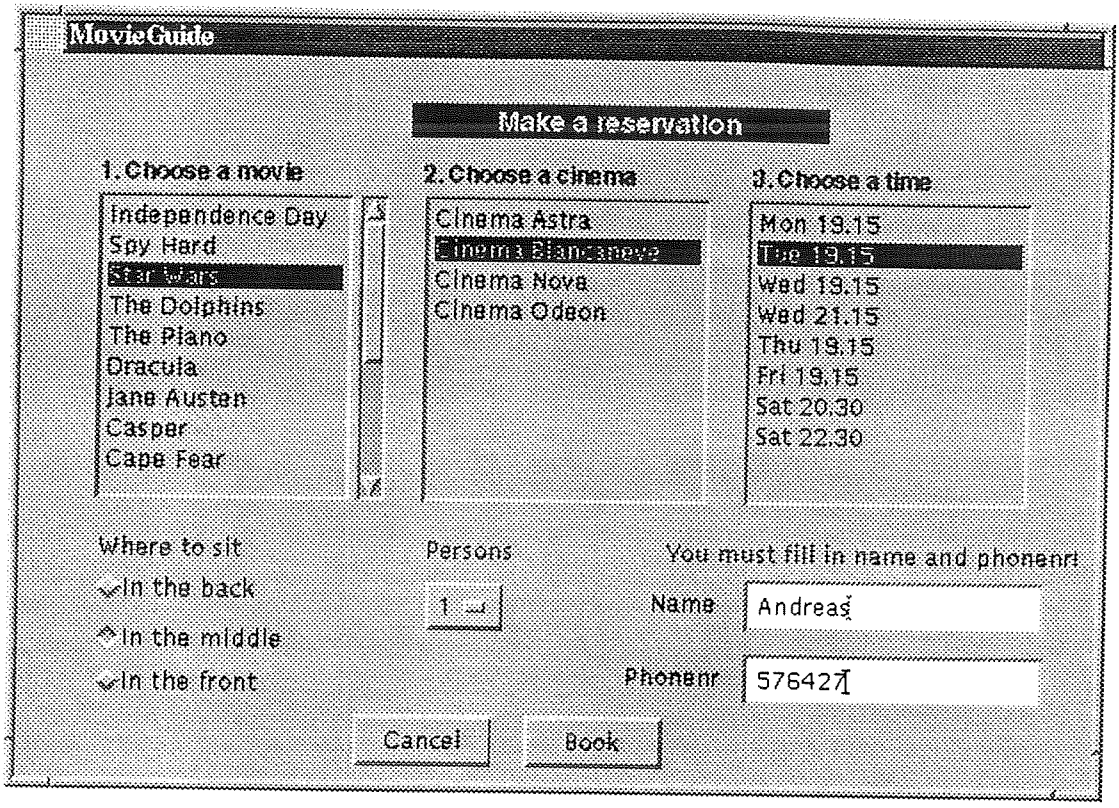


Figure 24 The new version of the reservation part, after the evaluation

6. Conclusions and future work

The method we have proposed has some advantages. One is derived from the use of a task model that allows the designer to specify a rich set of temporal relationships among tasks. The association between user actions and tasks can be used to see to which task the current error belongs. This makes it possible to find the tasks causing the user problems, and therefore to find the parts of the user interface that must be improved.

The development of the task model of an existing system forces the designer to think of how different parts of the user interface are developed and it is useful to analyse the design choices.

Another advantage is that the results of our method give the evaluator a possibility to *measure* the usability of the current application. This can be done by specifying usability goals. However, the goals must be of the kind "the time to perform some tasks must not be more than 2 minutes", or "the user must only do 2 errors on this task". If the evaluator set up these kinds of goals it is possible to see if they were satisfied or not after a user test, something that is hard to do from only observational data.

The goal of this work was to find a method to evaluate a user interface that finds the tasks and the errors performed during a user test, and the reasons for the major part of the errors. In addition, we have found a method to get the preconditions from the task model and to get the available tasks from the current state of the user session. The latter is not only useful for evaluation purposes but also to supply built-in task oriented help.

44

As a complement to user testing our method can give the evaluator more specified and concrete results as the tasks and errors performed. The combination with user testing gives additional information useful for the evaluation.

The proposed method does not though give explicit recommendations for the evaluator how to improve the user interface. However, our method can give support when deciding which improvements must be done and where in the interface they should be done. This is possible due to the precondition errors pointing out those tasks that caused the user problems. This means that our method could be a valuable complement and facility to usability testing.

The definition of usability includes relevance, efficiency, attitude, learnability and safety. The proposed evaluation method includes the possibility to measure efficiency and to a certain degree learnability and relevance. The learnability can e.g. be measured by the time completion rate and the numbers of errors. The relevance can be measured by examining if the user succeeded to fulfil the desired tasks. However, we do not cover attitude and safety yet. Future work could thus concern to cover more aspects of usability.

References

Bevan N. (1995). Measuring usability as quality of use. *Software Quality Journal 4*, pp 115-130. Chapman & Hall.

Brown C.M. (1988). *Human-computer interface design guidelines.* Norwood, N.J.: Ablex Publishing Corporation.

Card S.K., Moran T.P. and Newell A. (1983). *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Diaper D. (1989). *Task Analysis for Human-Computer Interaction.* Chichester: Ellis Horwood.

Dumas J.S. and Redish J.C. (1994). *A practical guide to usability testing.* Norwood, New Jersey: Ablex publishing corporation.

Good M., Spine T.M., Whitside J. and George P. (1986). User-derived impact analysis as a tool for usability engineering. In *Human Factors in Computing Systems (CHI'86 Conference Proceedings)*, Mantei M. and Oberton P. (eds.), pp 241-246. New York: ACM Press.

Hamilton F. (1996). Predictive evaluation using Task Knowledge Structures. In *Human Factors in Computing Systems, CHI'96 Conference Companion*, pp. 261-262. Vancouver: ACM Press.

Hix D. and Hartson H.R. (1993). *Developing User Interfaces: Ensuring Usability Through Product and Process.* New York: John Wiley.

ISO (1988). Information Processing Systems - Open Systems Interconnection - LOTOS - *A Formal Description Based on Temporal Ordering of Observational Behaviour.* ISO/IS 8807. ISO Central Secretariat.

Jeffries R., Miller J.R., Wharton C. and Uyeda K.M. (1991). User interface evaluation in the real world: A comparison of four techniques. In *Human Factors in Computing Systems (CHI'91 Conference Proceedings)*, pp. 119-124. New York: ACM Press.

Kieras D.E., Wood S.D., Abotel K. and Hornof A. (1995). GLEAN: A Computer-Based Tool for Rapid GOMS Model, Usability Evaluation of User Interface Designs. *Proceedings of UIST'95*, pp. 91-100. New York: ACM Press.

Kirakowski J. and Corbett M. (1993). SUMI: the Software Usability Measurement Inventory, *British Journal of Educational Technology*, **24**(3) pp. 210-314.

Lewis C. and Rieman J. (1993). *Task-Centered User Interface Design: A practical introduction.* A shareware book published on the web by the authors: www2.umassd.edu/CoursePages/HCI/hcireadings/TextVersion/index.htm. Also available at ftp.cs.colorado.edu/pub/cs/distribs/clewis/HCI-Design-Book

Löwgren J. (1993). *Human-computer interaction: What every system developer should know.* Lund: Studentlitteratur.

Löwgren J. (1995). Perspectives on Usability. *IDA Technical Report.* LiTH-IDA-R-95-23. ISSN-0281-4250. Department of Computer and Information Science, Linköping University, Sweden.

Macleod M., Bowden R. and Bevan N. (1994). The MUSiC Performance Measurement Method. In HCI '96, Tutorial 14, *Measuring Usability - MUSiC Methods,* (Bevan N.). London: The British HCI Group.

Macleod M., and Rengger R. (1993). The development of DRUM: a software tool for video-assisted usability evaluation. In: JL Alty et al. (Eds.) *People and Computers VIII (Proc. of HCI'93 Conf., Loughborough UK).* Cambridge: CUP, 293-309.

Nielsen J. (1993). *Usability Engineering.* Boston: Academic Press.

Pangoli S. and Paterno' F. (1995). Automatic Generation of Task-oriented Help. *Proceedings ACM Symposium on User Interfaces Software and Technology.* Pittsburgh: ACM Press.

Paterno' F., Mancini C. and Meniconi S. (1996). Understanding Tasks and Software Architecture Relationships, CNUCE Internal Report, December, *CNUCE-C.N.R.* Pisa, Italy.

Paterno' F., Mancini C. and Meniconi S. (1997). ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. Proceedings Interact'97, Chapman&Hall, Sydney.

Paterno' F., Sciacchitano M.S. and Löwgren J. (1995). A User Interface Evaluation Mapping Physical User Actions to Task-Driven Formal Specifications. In *Design, Specification and Verification of Interactive Systems'95. Proceedings of the Eurographics Workshop,* Palanque P. and Bastide R. (eds.), pp. 35-53. Toulouse: Springer Verlag.

Polson P.G., Lewis C., Rieman J. and Wharton C. (1992). Cognitive walkthroughs: A method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies,* 36, pp. 741-773.

Preece J. (ed.), Rogers Y., Sharp H., Benyon D., Holland S. and Carey T. (1994). *Human-Computer Interaction.* Workingham, England: Addison-Wesley.

Sears A. (1993). Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout. IEEE Transactions on Software Engineering, Vol.17, N.7, July 1993, pp.707-719.

Sears A. (1995). AIDE: A Step toward metric-based interface development tools. *Proceedings of UIST'95,* pp. 101-110. New York: ACM Press.

Wharton, C., Rieman, J., Lewis, C. and Polson, P. (1994). The cognitive walkthrough: A practitioner's guide. In *Usability Inspection Methods,* Nielsen J. and Mack R.L. (eds.). New York: John Wiley & Sons.

Wilson S., Johnson P., Kelly C., Cunningham J. and Markopoulos P. (1993). Beyond Hacking: A Model-based Approach to User Interface Design. *Proceedings HCI'93.* In: JL Alty et al. (Eds.) *People and Computers VIII (Proc. of HCI'93 Conf., Loughborough UK).* Cambridge: CUP

46