Automatic Support for Usability Evaluation

Andreas Lecerof and Fabio Paternò

Abstract—The main goal of this work is to propose a method to evaluate user interfaces using task models and logs generated from a user test of an application. The method can be incorporated into an automatic tool which gives the designer information useful to evaluate and improve the user interface. These results include an analysis of the tasks which have been accomplished, those which failed and those never tried, user errors and their type, time-related information, task patterns among the accomplished tasks, and the available tasks from the current state of the user session. This information is also useful to an evaluator checking whether the specified usability goals have been accomplished.

Index Terms—User interfaces, usability engineering, formal methods for human-computer interaction, task models, user interface evaluation.

1 Introduction

Interest in human factors in connection with 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 good usability.

Usability engineering is an area in HCI which aims to achieve usable systems by applying different methods at different stages of the design and development process in a structured and systematic manner. In the early stages of the design process usability evaluation is used to sort out alternative user interface designs and then to identify the preferred design. Later on, evaluation is performed to determine whether 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 [20]:

- 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?

There are several methods which have proved to be useful in supporting usability engineering. In *heuristic evaluation* [14] user interface designers study the interface and look for properties that they know lead to usability problems. Evaluators with good experience are likely to find many problems with this approach although non-experts too may also find some problems. The main problem is that usually the list of identified problems is useful to improve the current design but it may overlook other relevant usability problems.

Another approach is *usability testing* [4]. In this case the designer studies the user performing some tasks and gathers data on the problems that arise. The problems with usability testing is that it is based only on observational data and to be able to interpret the data some user interface experience is needed and some problems may still not be identified. There is also the problem of cost, the time of the users and the observers is taken, and it may be difficult to get the appropriate users for the test.

In this work we want to show how automatic support can be given in usability evaluation in such a way to have more complete and reliable results which can reinforce and complement those obtained by other methods such as user testing.

We propose a method of evaluating a user interface with the use of a task model and the logs generated from user tests of the application considered. The information from the analysis should then be used to improve the usability of the application. The method we propose is intended to be carried out by designers. We try to combine empirical testing with the information from the task model of the application. The aim is to give the designer information on how to improve the interface.

More specifically, the goals of this work are:

- *To find a method of evaluating a user interface using task models and logs from user tests of an application*

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

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

• *A. Lecerof is with CNUCE-CNR, Via S. Maria 36, 56126 Pisa Italy, and the Department of Computer and Information Science, Linköping University, Sweden. E-mail: andle@iname.com.*
• *F. Paternò is with CNUCE-CNR, Via S. Maria 36, 56126 Pisa Italy. E-mail: f.paterno@cnuce.cnr.it.*

Manuscript received 13 May 1997; revised 28 Apr. 1998.
Recommended for acceptance by J. Gannon.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105050.

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

The paper is organized as follows: Section 2 introduces the possible results of usability engineering and user interface evaluation, and discusses and compares some evaluation approaches. Section 3 explains the notation and the method of building task models used in this work and how it can be used for user interface evaluation. Section 4 describes the proposed method for evaluation and its results. It also provides an example of use. Section 5 describes the tools used in this work, the tool to record the user logs and the USINE tool to execute the evaluation method. Section 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 discuss the costs and benefits of our method and we draw some conclusions from this work and provide some suggestions for future work.

2 Background

2.1 Basic Concepts in Usability Engineering

The main goal of *usability engineering* [14] is to improve the user interface. Usability is concerned with making systems easy to use and easy to learn. But this is not all. For example, if users cannot carry out all the tasks they want, because a feature is missing in the system, it is not likely they will agree that the system is usable. A broader definition is therefore preferable, 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 the 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 factors, 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 and other similar factors can be the result of an evaluation of a user interface. Some of the benefits of having measurable terms are:

- 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 achieve these goals.
- It will be easier to compare requirements of alternative designs and how much more work is required to improve them though it is not possible to convert exactly the measurable goals into the amount of time necessary to achieve them.

2.2 User Interface Evaluation

When evaluating a user interface, it is important to know what usability means for the current application. For example, a flight control program must have certain features that a word processor does not have to 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 cannot know which tasks s/he must be able to perform. 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 on identifying usability problems and improving the user interface. This is the focus we use in this work.

It is important to consider task-related aspects in the evaluation of the user interface. A task is an activity performed to reach a goal. We can think of tasks at different abstraction levels, ranging from high level tasks (such as *retrieving information on film projections available today*) to very low level tasks (such as *selecting a button on the screen*). In both cases we can think of tasks which can or cannot be performed and of temporal ordering among tasks. More specifically, in case of high level tasks the temporal relationships are determined by the logical dependencies among tasks (for example printing a file can be performed only after indicating the name of the file to print) whereas in the case of the low level tasks temporal relationships may depend also on constraints provided by the implementation of the user interface.

Generally speaking an evaluation of a user interface after performing a user test should include the following:

The tasks the user was able to perform

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

The tasks the user was not able 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 its performance should be supported efficiently (for example, designers can decide to provide shortcuts or macros).

In which order the tasks were performed

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

The different errors the user made

The user errors can be of different types. If the user makes an unnecessary action in performing the current task this is, in most cases, an error. However, the user may have 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 track before continuing. For example, before the user decides to delete some selected files s/he may want to be sure that s/he has made a correct selection, i.e., so only the files meant to be deleted are selected.

A common type of error is an action performed belonging to the task, but the user has failed to do some actions needed before. For example, if the user tries to print a file without specifying the name of the file. In this case the user performed the right action (pushing the print button) but the *precondition* of the task (specifying the name of the file) was not satisfied. An error could also arise if the user inputs something 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 as to which part of the user interface s/he liked or disliked.

How much help the user needed

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

How many times the user had to restart from the beginning

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

How long it took the user to perform the tasks

If time is an important factor for users of an application, which is often the case, then it is necessary to measure the time required to perform tasks by the current design.

This is a nonexhaustive list of information which should suggest to the designer what improvements could be made to the user interface.

2.3 Related Works

The *MUSiC performance measurement method* [13] was developed by the European MUSiC (**M**etrics for **U**sability **S**tandards in **C**omputing) project to provide a valid and reliable means of specifying and measuring usability. The method gives useful feedback on how to improve the usability of the design. MUSiC also includes tools and techniques for measuring user performance and satisfaction.

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 provide valuable data about specific areas where the design fails to support the users' performance. The method provides suggestions for 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 helps to identify where improvements to the user interface have to be made.

Unlike the MUSiC method, *cognitive walkthrough* [23], [12] can be performed 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 first to 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. 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 to do in their work. Then a relevant task that the design is intended to support is chosen. After this you try to tell a believable story about each step and action the user has to do to accomplish the task. To make the story believable you have to motivate every action the user performs, based on knowledge about the user and the feedback the user gets from the system. If you cannot tell a believable story about an action, you have found a problem with the interface. Before performing a cognitive walkthrough you must have the following: 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 accomplish the task; knowledge about the user and his/her work tasks.

There are other inspection-based evaluation techniques. For example, think aloud evaluation [11] where the user during the test is continuously thinking out loud thus allowing the evaluator to better understand when and why s/he has problems with the interface. All of these techniques require the usability specialist to be continuously present and actively involved in every evaluation. This can be very expensive, especially for large tests. Our proposal (USINE) aims to decrease such involvement, while at the same time allowing evaluators to have a good understanding of real user behavior in the workplace through the support of automatic tools when they have to improve the user interface design.

Jeffries et al. [8] 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 especially important that the user can understand and use the interface quickly and easily, 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 that you are not likely to find all the existing problems with the user interface by this method, though we should say that no method guarantees all usability problems will be found.

Empirical testing, e.g., testing used in the MUSiC method, consists of testing performed by end users and collecting related data. It is often used in iterative design revisions where real users test and help to find usability problems. The involvement of real users makes it more expensive but at the same time more reliable, at least from the users' point of view. The major disadvantage with this method is that it is expensive. This is because you use the time of the users, thus preventing them from carrying out their *real* work. It is also a problem finding the necessary numbers of users who are both domain experts and who belong to the target group. Because of this, "discount methods" [14] and "inspection methods" (e.g., cognitive walkthrough) have been developed, for assessing the usability of an interface design very quickly and at a lower cost. The lower cost is due to the fact that you do not use real users in the methods.

Another possibility is *model-based approaches* to usability evaluation. They often aim 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, [2]) to find the applicable usability problems of the interface. A GOMS model is a representation of the *procedural knowledge* users 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 precise, down to keystrokes. Therefore, it is associated with "keystroke-level analysis." This method makes it possible to predict the time needed to complete a task within a 20 percent margin [12]. Works such as GLEAN [9] 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) develops a GOMS model for an existing or proposed interface and supply a representative task and a description of the interface behavior. GLEAN then simulates the user interaction and generates usability parameters such as the learning time for the task. The most important factor of GLEAN is that it automates the tedious calculations required to generate usability predictions from the GOMS model. 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. An evolution of this approach is EPIC (Executive-Process/Interactive Control) [10] which provides a framework for constructing models of human-computer interaction by taking into account results on human perceptual/motor performance, cognitive modeling techniques, and task analysis methodology implemented in the form of computer simulation software. Then by running the model in a simulated interaction they can obtain statistics that predict human performance with the actual system.

AIDE [22] 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 [21] 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 contrast 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 perform their tasks, and not only the efficiency of the layout.

In [18] a user interface is evaluated using the architectural specification and 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 to find out how the user interface can be improved.

Approaches such as GLEAN are methods best used early in the design to predict the execution time. It is not easy to use for non-experts. Another disadvantage of models like GLEAN, is that they presume an error-free behavior of the user when performing the task. This means that you do not take user errors into 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 do not directly involve users in the evaluation. We believe that involving users can give better information on how to improve the usability of the user interface. Unlike GLEAN our approach also takes into account the user errors, because we think they can give important information, guiding the designer on how to improve the interface. We also want our method to be easy, so that the designer does not have to spend most 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.

More specifically, the aim of our method is to overcome some of the limitations of the methods described in this section: 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 proposed method shows how an analysis of the tasks and the errors can give suggestions on how to improve the interface.

3 Task Models

This section describes the notation and the approach we used for building task models in this work. We also introduce how it can be used for user interface evaluation.

Task models can give useful information for evaluating user interfaces. Task analysis [3] usually begins with investigating the users' current problem and breaking down the tasks that potential users of the system do or could do. A task model describes the set of activities required to reach the users' goals and how these activities are related to each other. A goal is a state of an application that the user wishes to achieve or the access to an application to get information. We will use the task model to describe both logical activities and the use of the interaction techniques supported by existing user interfaces. This means we will refine the task model specification in more detail.

Task models have been used to develop interactive applications. Methods such as TLIM (**T**ask **L**OTOS **I**nteractor **M**odeling) [17] and Adept [24] 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 in our view they can also give useful support to engineering the usability evaluation phase.

The notation for specifying the task model we use is *ConcurTaskTrees* [16]: this notation allows designers to specify the temporal relationships among tasks and other information, useful for the design of the user interface. It is supported by a graphical editor (http://giove.cnuce.cnr.it/ctte.html) for creating the task tree and specifying relationships among tasks according to its syntax and semantics (see Fig. 1).

Only the parts of the TLIM method and the ConcurTaskTrees notation of interest for this work are described here. For example, the TLIM method 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 [17]).

3.1 The ConcurTaskTrees Notation

A task model is useful to understand what the current intention of the user is. In ConcurTaskTrees the task model is represented by a hierarchy of tasks where higher levels are more abstract and logical and lower levels are more refined and concrete and oriented to describe the physical actions required to interact with the user interface. The leaves in

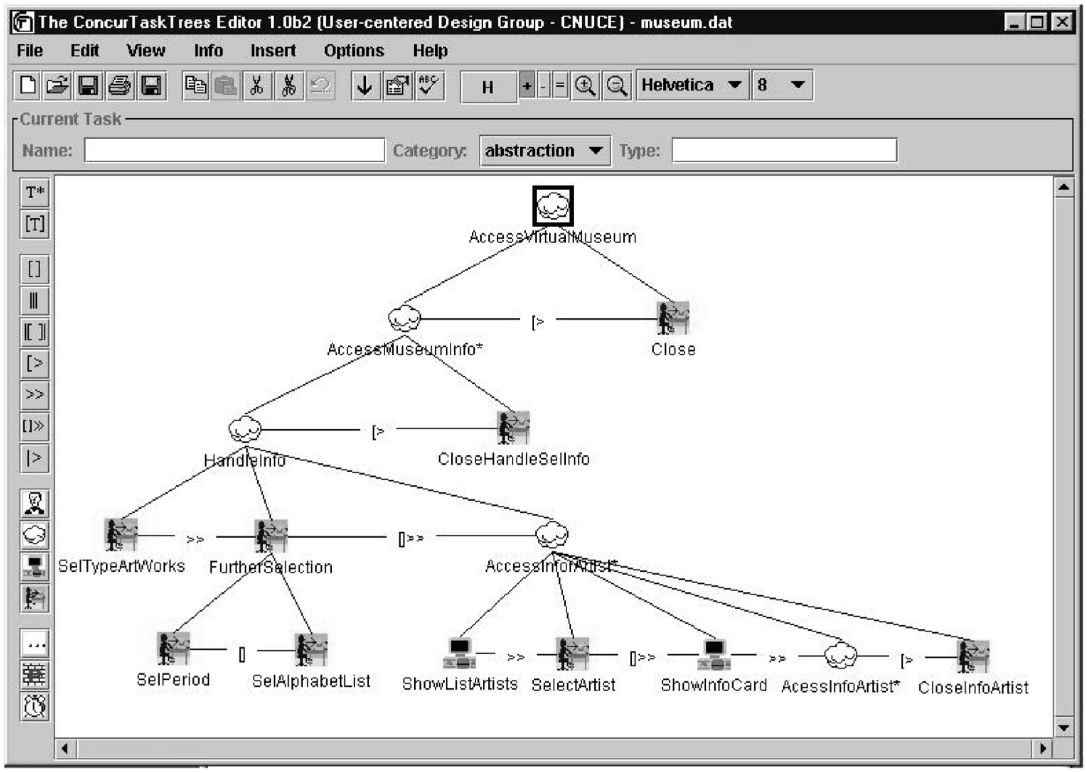


Fig. 1. The ConcurTaskTrees model editor.

the task tree, the tasks which are not further decomposed, are called *basic tasks*.

In the ConcurTaskTrees notation there are four categories of tasks (see Fig. 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 application, e.g., presenting results given by 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 been decided, e.g., a user session with a system.

The task model in ConcurTaskTrees is built in three phases:

- Firstly, we make a hierarchical and logical decomposition of the tasks represented in a tree structure. High level tasks are decomposed into subtasks which describe more precisely possible activities.
- Secondly, we identify the temporal relationships between the tasks.
- Then, we identify the objects associated with each task. Objects are things which are manipulated to perform the task, e.g., the object of the *Send a letter* task is the letter itself. Finally, actions which describe how these objects can communicate with each other are also considered. This part of the notation (the specification of objects and actions) is not used in the work presented in this paper.

3.1.1 The Temporal Relationships Among Tasks

The temporal relationships between the tasks are expressed by extending operators of the LOTOS notation [7]. ConcurTaskTrees allows designers to describe concurrent tasks unlike GOMS [2] which can only analyse sequential tasks.

The main operators we use are:

- T1 ||| T2, *interleaving,* task T1 and task T2 can be performed in any order.
- T1 |[]| T2, *synchronization,* the two tasks T1 and T2 have to synchronize, i.e., they have to perform some actions at the same time, to exchange information.
- T1 >> T2, *enabling,* the performance of task T1 makes it possible to perform T2. Task T2 cannot be performed 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 once.

We can also specify *optional tasks*, tasks the user decides whether 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 number of copies].

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

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

To solve the ambiguity problem 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 Fig. 4.

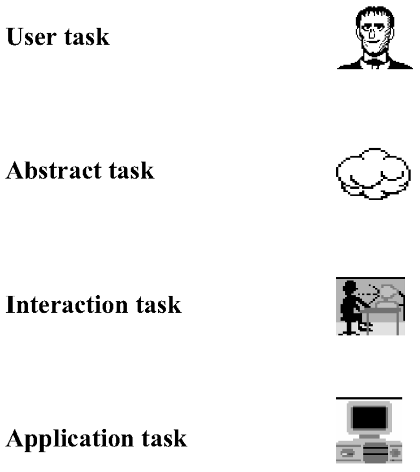


Fig. 2. Graphic representations of task categories.

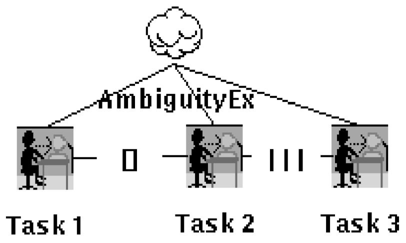


Fig. 3. An example of ambiguity.

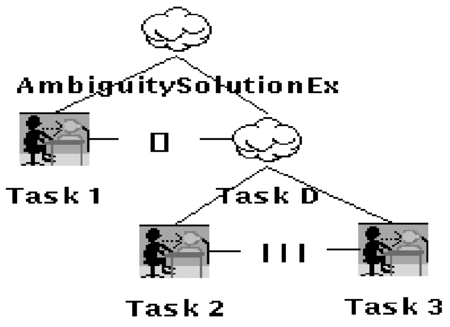


Fig. 4. A solution to solve ambiguity.

3.1.2 A Small Example of a Task Model Using ConcurTaskTrees

In the tree in Fig. 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 else. The *Choose file* task enables the *Print* task. This means that the task *Print* gets the necessary information from the *Choose file* task to be available to the user.

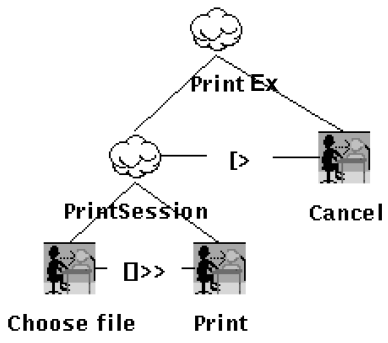


Fig. 5. A small example of a task tree.

3.2 How the Task Model can be Used for Evaluation

Task models are often used in the development and design of applications. In this work however we use them for a different purpose: to evaluate a user interface. This means that we will use tasks *differently*. Our interest is concentrated on interaction tasks because we want to evaluate the user interactions. However, we use other category of tasks as well, e.g., *abstract* task to specify complex tasks.

> *When we design the task model for evaluation purposes we specify the tasks in more detail until we obtain interaction basic tasks which are tasks which require one single user action to be performed.*

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

When the user performs a task s/he may change the set of available tasks to perform according to indications given by the model represented by the task tree. By analyzing the task tree and considering the tasks the user has performed, it is possible to know which tasks at each time the user can perform. These tasks, *the available tasks*, can be found due to the specification of 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 ConcurTaskTrees specification, e.g., when the precondition of a task is not satisfied.

The context-sensitive information given by the task tree is not only useful for evaluating a user interface but also for generating task-oriented contextual help [15].

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 application, 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

This section describes the proposed method for evaluating a user interface and the results achieved.

4.1 The Input

Fig. 6 shows the structure of the method. The designer creates the task model with the ConcurTaskTrees editor (see Section 3) and the log-task table with the USINE tool (see Section 5) that has been developed to provide automatic support for usability evaluation. This table allows designers to map physical actions performed by the user onto basic tasks of the task model. The user logs are provided by the log recording tool, Replay (see Section 5), from a user test of the current application. One or a few logs are sufficient to build the log-task table associated with one application whereas many more logs are used for evaluating the application.

The precondition table is then created automatically from the task model. The evaluation method finds the available tasks in the current state of the user session which is indicated by the actions stored in the log file, and at the end of the analysis of the session provides the results from the evaluation.

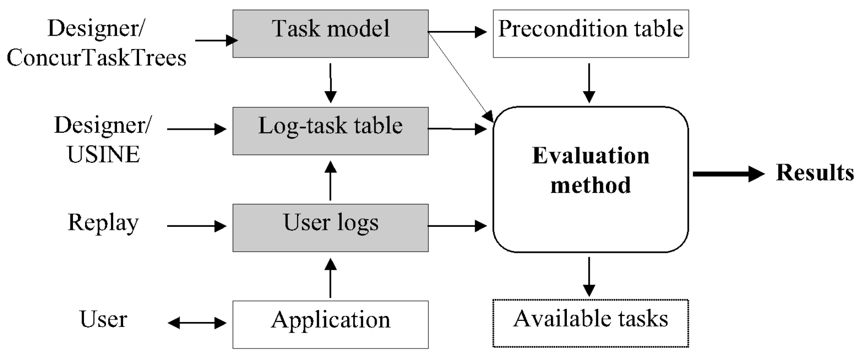


Fig. 6. Overview of the method.

From Fig. 6 it can be seen that as input our method requires four things:

1) Log files generated by user sessions with the application considered
2) The task model of the application
3) The log-task table, created with information from the task model and one log file
4) The precondition table, automatically created from the task model

4.1.1 The Logs

We got the logs from user tests by recording the actions performed by the user with the Replay tool (see Section 5). However other logging tools (such as JavaStar) can be used by requiring very minimal modifications to our environment. The logs are simple text files. Before we can use the logs we have to clean them from redundant information. For example, Replay creates logs such as "popup" and "popdown" indicating that a window is visible or not. These logs make it possible to replay the user session, however we do not use this feature.

4.1.2 The Task Model

The task model, as described in Section 3, is created by using the ConcurTaskTrees editor. We use the task model in order to specify the temporal relationships among tasks, i.e., the enabling and disabling tasks. The hierarchical task tree is decomposed so detailed that every basic task is completed by one action (one log) in the log file.

4.1.3 The Log-Task Table

In the log-task table, logs and tasks are associated by using the USINE tool. 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 Fig. 7).

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

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

The first column contains the physical user actions in the log file. The actions specified in the log-task table consist of the *type* of action and the *target* of the action. For example the type could 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 **basic task** that the action *executes*. That is, when the action is performed the task is accomplished. Thus we associate each basic task with a corresponding action. If the task is not an interaction basic task we put a "#" instead of the log.

In Fig. 7 "click {/Vote}" executes the *Send vote* task. The *Session* task is an abstract task and it is thus associated with a "#".

4.1.4 The Precondition Table

In the task model of the application it is possible to see the tasks enabling and disabling other tasks (see Section 3). From the task model we automatically create a table of preconditions, the *preconditions table*, used internally in the method. In the preconditions 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 completed. Each precondition is associated with a Boolean value indicating whether or not it is verified.

Thus, the notation used is:

```
<task>| precondition1<Boolean value>|
precondition2<Boolean value>
```

At the beginning of the session the Boolean variables have the false value:

```
<task>| precondition1<false>|
precondition2<false>
```

When the user performs a precondition we change the value of the Boolean variable associated with *true*, indicating that the precondition has been performed. Thus, for example, if precondition1 is verified we have:

```
<task>| precondition1<true>|
precondition2<false>
```

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

```
Send vote | Write your name<false>
```

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

```
Send vote | Write your name<true>
```

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

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

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

```
Select/insert data|Choose cinema<false>|
Write a name<true>|Write phonenr<false>
```

If the user undoes a precondition, e.g., unselects an object previously selected then the precondition in the table becomes false again.

If the precondition for a task is one task which can be chosen from multiple tasks we insert one line for each task which can be chosen and when one of these tasks is performed the related line is set to true. In this case, in order to check whether the precondition of the task considered is verified, it is sufficient to perform a logical OR among the tasks which can be precondition. For example, if either the *Choice director* or the *Choice category* must be performed before performing *Select a movie* then in the precondition table we have at the beginning of the session:

```
Select a movie|Choice director<false>
Select a movie|Choice category<false>
```

And if, for example, the *Choice director* task is performed then the table is updated in the following way:

```
Select a movie|Choice director<true>
Select a movie|Choice category<false>
```

This will be sufficient to consider the precondition for the *Select a movie* task verified as the logical OR among the tasks which can be preconditions now gives true as a result.

4.2 The Method for Getting the Preconditions from the Task Model

The method for finding the preconditions and creating the precondition table searches through a preorder traversal of the task tree. For every nonoptional task we check if its ***left brother is an enabling task***. If so we know that the left brother is the precondition of the current task and write this (the task and its precondition) to the result.

If the current task is abstract then the left enabling brother is a precondition also for the children of the abstract task that are available at the beginning of its performance. The children available at the beginning of the performance of the parent task are those which are on the left of the leftmost enabling operator. If there is no enabling operator it means that all the children are available at the beginning.

In any case when we consider an abstract task we have to search for its preconditions which are among its children. While searching the method collects the results.

The method is thus as described above and divided into two parts. The main steps of the method are described in the upper part of Fig. 8:

- If the current task has a left enabling brother, the left brother is the precondition of the current task. For example (see Fig. 9), if the current task is *Login* we will find that it has a left enabling brother, *WritePassword*. This means that *WritePassword* is the precondition of *Login*.
- If the current task that has a left enabling brother is an abstract task, then also its children that are available at the beginning of its performance have the left enabling brother as a precondition. This is the case of *LoginDialog* in Fig. 9 with the abstract left enabling brother *PersonInfo*. We will first put *PersonInfo* as the precondition of *LoginDialog*. Then we find also that *PersonInfo* is a precondition for *WritePassword*.

We can have multiple tasks which share the same left enabling task which thus have the same precondition.

The description of the step concerning abstract tasks is expanded in the lower part of Fig. 8. The method of getting the preconditions of an ***abstract task*** searches its children. The method starts with testing whether all children are checked. It has the following characteristics:

- If the current task (a child of an abstract task) is an optional task, because of its type, the task does not need to be performed, it is not counted as a needed precondition and thus is not included in the results.
- If there are tasks composed by the choice operator ("[]") or the disabling operator ("[>") we will add these tasks as a precondition to their parent but on different lines (the performance of only one of them is sufficient to satisfy the precondition).
- If the current task is the rightmost and has the enabling operator on the left we will add this task as a precondition to its parent. This is because in this case the current task has another precondition that must be performed before, and we only want the strictly needed preconditions of the parent of the current task.
- If the current task is on the left of the rightmost enabling and has a right brother that is optional we must select the current task as a precondition to its parent. This is because we do not know if the optional task will be performed or not.
- If the children of the abstract task are composed by the interleaving operator (|||) then we will add them as a precondition. When all the children have been performed then this precondition is verified.

More specifically, if we apply the algorithm to the example in Fig. 9 we will find that *LoginSession* and *Quit* are preconditions for the *LoginExample* task, since they are composed by the disabling operator, the performance of one of them is a sufficient precondition for terminating the *LoginExample* task. Then we consider the subtasks of the *LoginSession* tasks, there is an enabling operator and thus we can find first that *PersonInfo* is a precondition of *LoginDialog* and next that it is a precondition of *WritePassword* too. Next, the method will find *Write name* and *Write phonenr,* which are composed by the interleaving operator, as preconditions for the *PersonInfo* task. Finally, we can find that *Login* has *WritePassword,* which is a left enabling brother, as precondition, and *Login* is precondition for *LoginDialogue,* in addition to *PersonInfo* which was found before.

In the preconditions table we will write the following:

```
Login example    |Quit
Login example    |Login Session
LoginDialog      |PersonInfo|Login
PersonInfo       |Write name
                 |Write phonenr
LoginSession     |LoginDialog
WritePassword    |PersonInfo
Login            |WritePassword
```

4.3 The Available Tasks

When the user has succeeded in accomplishing a task the precondition table is updated and it is possible to get the updated list of available tasks from it by identifying the tasks which have the precondition satisfied with the addition of those which have no precondition and so they are available at any time. The available tasks are those the user can perform right after s/he has performed the current task. The evaluator can use the available tasks to understand the user's preferences, within the alternatives provided by the system, when performing a task.

When we are analyzing the task tree to identify the available tasks we are only interested in the *interaction* tasks, which are those the user performs when interacting with the system. The method for getting the available tasks is used after 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.

4.4 The Evaluation Method

The evaluation process starts by considering the first action in the log file. We first get an action from the log file and if the action exists in the log-task table then we look up the

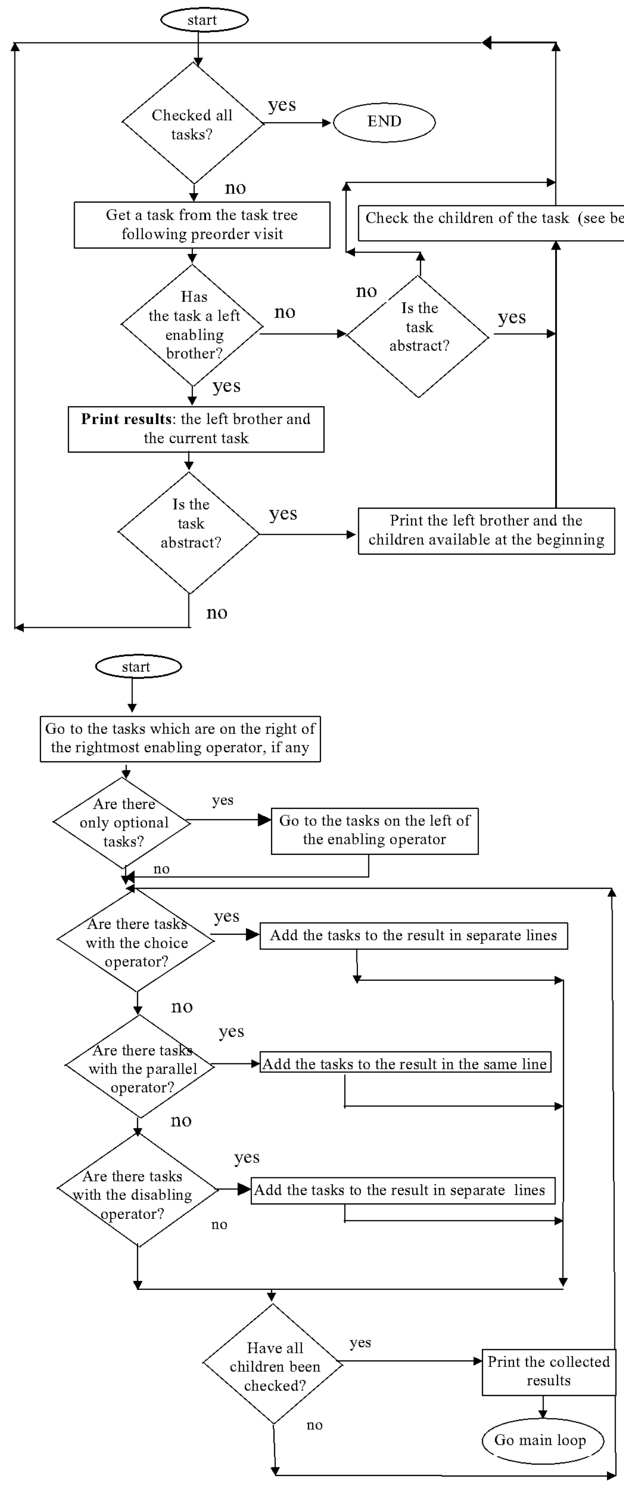

start
Checked all tasks?
yes
END
no
Get a task from the task tree following preorder visit
Check the children of the task (see below)
Has the task a left enabling brother?
no
Is the task abstract?
yes
yes
Print results: the left brother and the current task
Is the task abstract?
yes
Print the left brother and the children available at the beginning
no
start
Go to the tasks which are on the right of the rightmost enabling operator, if any
Are there only optional tasks?
yes
Go to the tasks on the left of the enabling operator
no
Are there tasks with the choice operator?
yes
Add the tasks to the result in separate lines
no
Are there tasks with the parallel operator?
yes
Add the tasks to the result in the same line
no
Are there tasks with the disabling operator?
yes
Add the tasks to the result in separate lines
no
Have all children been checked?
yes
Print the collected results
no
Go main loop

Fig. 8. The method for getting the preconditions.

associated task, otherwise an error is generated. If the task has preconditions we then see if they are satisfied and this means that the task is accomplished. If the preconditions are not satisfied then we have a precondition error. If the accomplished task is a precondition of another task then we change the precondition table. Then we get the available tasks from the current state, and we move to consider the next action in the log file.

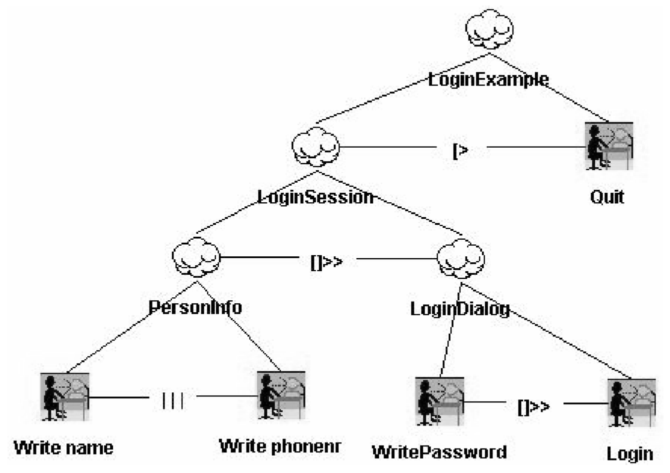


Fig. 9. An example of a tree with preconditions.

4.5 An Example of the Evaluation Method in Use

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

The related initial precondition table is the following:

```
EvaluateEx   |Close<false>
EvaluateEx   |Session<false>
Session      |Reservation<false>|Search<false>
Reservation  |Book<false>
Search       |Send query<false>
Book         |Insert Data<false>
Send query   |Insert criteria<false>
Insert Data  |Write name<false>
             |Choose  movie<false>
```

At the beginning of the session the immediately available tasks (those which have no precondition) are *Write name, Choose movie, Insert criteria,* and *Close.*

The first thing we do in the evaluation process is to read an action from the log file (in this example only the important part of the information in the log file is shown, the coordinates are omitted). Before the beginning of the user session all the preconditions are set to false. Below we show how the elements of the precondition table are modified according to the logs detected.

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

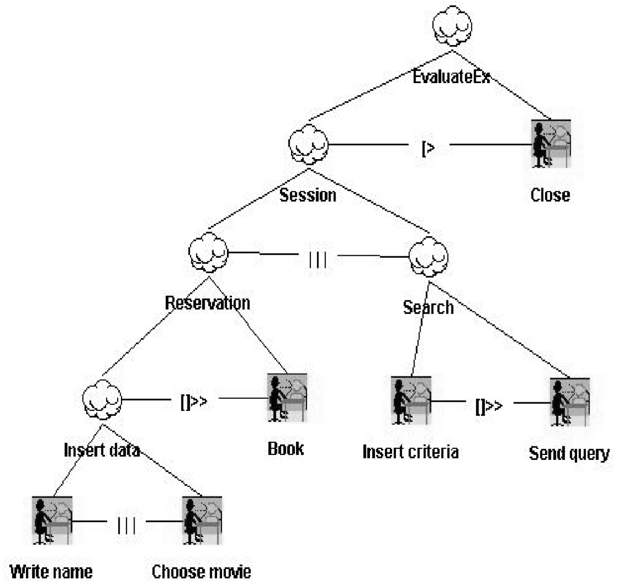
This action is not found in the log-task table, i.e., it is not associated with any task and we will record this as an error.

This error occurred because the designer did not specify this action 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 a function, e.g., things like images, labels, or text strings.

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

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

```
Send query|Insert criteria<false>
```



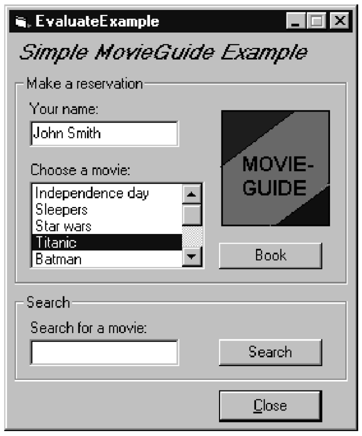


Fig. 10. The task tree of the example and the related user interface.

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. We will consider this user action as a **precondition error**, because the precondition of the task that the user tried to perform was not satisfied. That is, the user tried to perform the *Send query* task before s/he had inserted the search criteria.

```
3. text {\Insert criteria}
```

The next action is associated with the *Insert criteria* task (to insert a search criteria) in the log-task table. This task does not have any preconditions so the task is completed or in other words achieved. However, the *Insert criteria* task **is** itself a precondition of another task, *Send query*. This means that we must change in the preconditions table, the `Insert criteria` field to true:

```
Send query|Insert criteria<true>
```

After having updated the precondition table the tool can add *Send query* to the list of available tasks.

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

The next action is `click {\Send query}`. When we check the precondition table this time, because of the previous action, we will find that the precondition for this task now is satisfied. The user has now successfully performed the *Send query* task.

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

Action No. 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. However the task is a precondition *itself*. It is the precondition for the *Insert data* abstract task. Thus in the preconditions table we change the field of this precondition for *Insert data* to true:

```
Insert data|Write name<true>|Choose
movie<false>
```

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

The next action is associated with the *Choose movie* task. 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 related field in the precondition table:

```
Insert data|Write name<true>|Choose
movie<true>
```

The *Insert data* task is an abstract task and thus it cannot be performed by interaction from the user. In this case it is considered performed because both its subtasks are performed at this point of the session considered, and thus it is added to the list of available tasks.

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

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

4.6 The Results from the Evaluation

The results of our evaluation method concern how well users succeed in performing their tasks and the errors the users made. 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
- The user errors of two kinds
- Numerical and temporal information of the above. For example, we can find how many times the tasks and the errors were performed and the temporal order of the tasks and the errors.
- How long each task took to complete. This is calculated based on the time stamp that exists on every action, showing how much time passed since the last action was performed, information provided by the recording tool (Replay).
- The times for the abstract tasks 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 occurred at the beginning it could mean that the user should have more support and help at the beginning of the interaction.
- Task patterns, i.e., sequences of accomplished tasks that occur more than once. This information can be used to identify where macros should be implemented or where some automation should be performed.
- For example, if the *Choose printer* task always is followed by the *Print a file* task, the *Choose printer* task could be done or set automatically by the program.
- The available tasks from the current state of the user session, i.e., from a certain point in the log file.
- Other useful information, such as the test time, how many times the scrollbar was used and how many times the window was resized.

These results are elaborated by the USINE tool (see Section 5) where they are presented in different ways.

4.6.1 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.

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

The information about the tasks the user accomplished is useful when usability goals have been specified (as mentioned in Section 2). For example, if the goal is that the user must accomplish certain tasks, it is possible to see if the goal was satisfied after the evaluation. It is possible to view the results in the temporal order of the tasks, or ordered by task, in our tool USINE (see Section 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 require less mouse movements, thus making the interface more efficient [22].

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

4.6.2 The User Errors

The user errors are of two kinds. The first kind consists of actions needed to perform a task but the preconditions of the related task were not fulfilled (precondition errors) when they occurred. This type of error causes the failed tasks described in the previous paragraph. 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 allows us to conclude how *easy* the user can perform his/her tasks, and which tasks caused problems for the user. Errors are often a part of the usability goals [4], for example, a goal could be that the user should make no more than five 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. As a result the information from the evaluation method provides the necessary information to measure if such goals have been achieved.

4.6.3 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 Section 4 we use two tools, QC/Replay and USINE. QC/Replay is commercial software while USINE is the tool developed as part of this work.

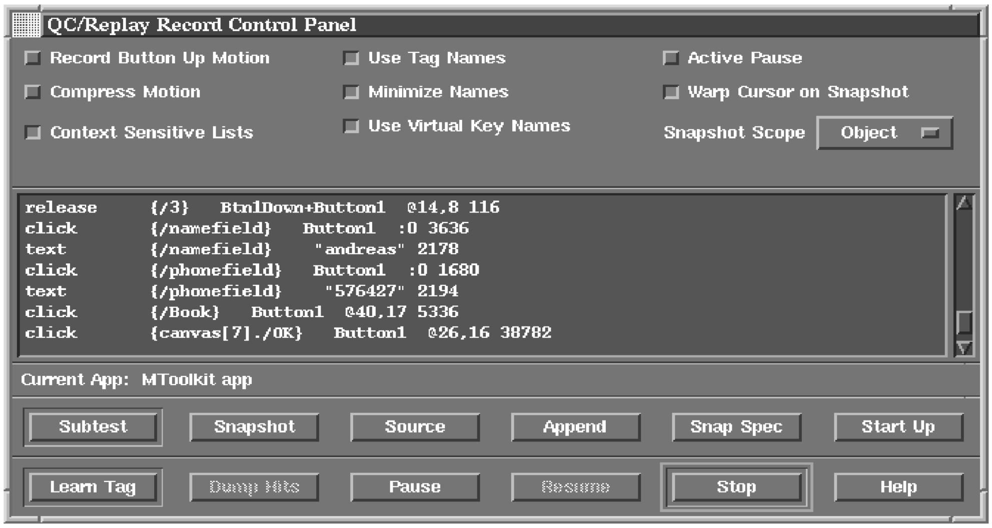
To record the actions from the user test we used QC/Replay (Replay) version 2.5 from CenterLine Software.[1] It works in X Window System environment and it can support any toolkit for user interfaces in this environment without requiring any specific programming.

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

The actions recorded by QC/Replay (see Fig. 11) include pressing and releasing the mouse, double-clicks and keys pressed. A typical line in the log file includes:

- The type of 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 in a list
- The coordinates of the mouse click
- The time elapsed from the previous log

1. Internet: http://www.centerline.com/productline/qcreplay/qcrplay.html



```
release     {/3}   Btn1Down+Button1  @14,8 116
click       {/namefield}   Button1  :0 3636
text        {/namefield}    "andreas" 2178
click       {/phonefield}   Button1  :0 1680
text        {/phonefield}    "576427" 2194
click       {/Book}   Button1  @40,17 5336
click       {canvas[7]./OK}   Button1  @26,16 38782
```

Current App: MToolkit app

Fig. 11. The QC/Replay record control panel.

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 required.

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
3) The file with the log-task table

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

5.1 The Preparation Part

The preparation part supports the creation of the log-task table as described in Section 4. The designer can load the tasks from the task model of the application on the right side and the log file from the user test on the left side (see Fig. 12). The tasks are indented depending on their level in the task tree, the further to the right the deeper the level.

The designer then associates each basic task with the action in the log file that executes it. One task can be performed a multiple number of times in one session but the association needs to be created only once. To this end we need a log file which contains all the actions associated with the tasks in the task model. This can be done directly by considering a session performed by somebody who knows the application well and who can rapidly perform all the tasks. Alternatively we can consider a normal user session and if there are some tasks which are not performed and some which are performed in other user sessions, our tool gives the possibility of updating the log-task table by using the *Add* button showed in Fig. 12 whereas the *Delete* button is useful for removing information inserted by the QC/Replay tool which is not relevant in our method.

The task model is decomposed in such a way as to have basic tasks corresponding to button selections, menu selections and similar basic interactions. Abstract, user, and application tasks, that do not have any corresponding action in the user interface, are marked in the log-task table with a “#”. Optional tasks (see Section 3) are marked within brackets “[]”.

It is also possible to load an existing log-task table to edit it. The *home* button in the user interface allows the evaluator to go to the initial presentation of the USINE tool where s/he can ask either to enter into the preparation part or the evaluation part.

5.2 The Result Part

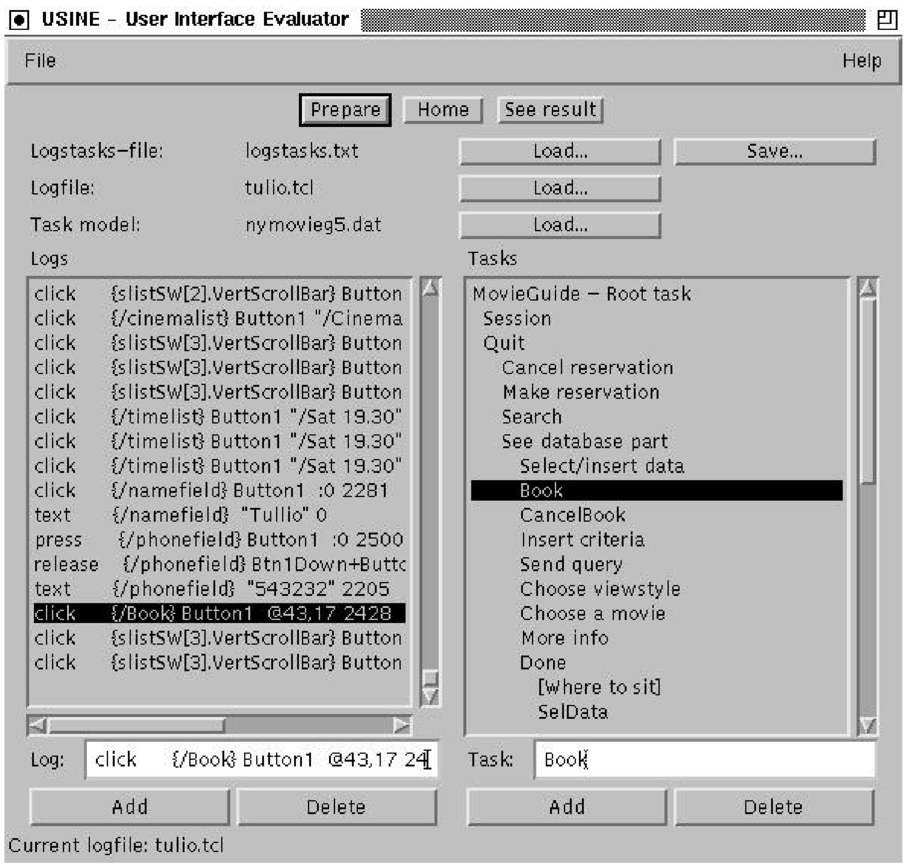
From the preparation part we can activate the result part by selecting the *See result* button (Fig. 12). In the result part the user starts the evaluation by pressing the *Evaluate* button (see Fig. 13). The evaluation method of the tool is the one described in Section 4. The evaluation is shown in the text area as it proceeds. The results shown are the user actions in temporal order and the type of the task or the associated error, depending on whether the user succeeded in accomplishing the task or not. If the user succeeded in accomplishing a task, the following available tasks, which can be performed in the next step, are shown.

The different formats used in USINE (see Fig. 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.,



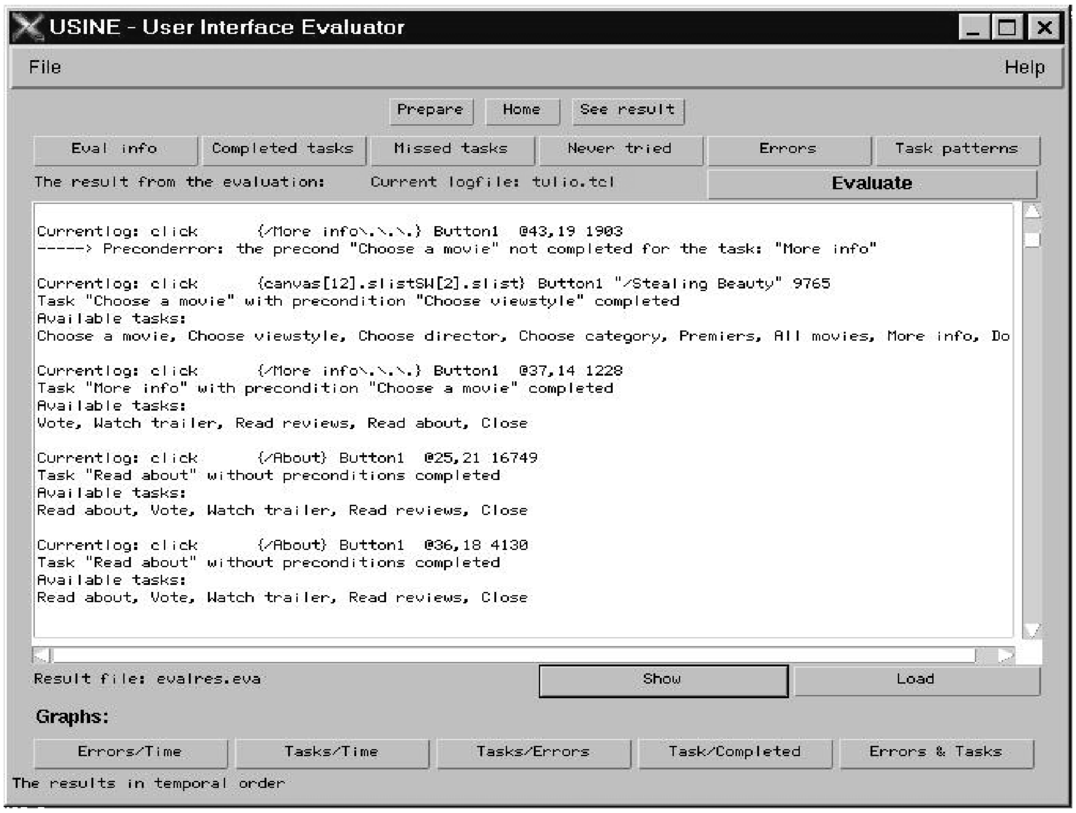
```
Logstasks-file:   logstasks.txt
Logfile:          tulio.tcl
Task model:       nymovieg5.dat
```

```
click     {slistSW[2].VertScrollBar} Button
click     {/cinemalist} Button1 "/Cinema
click     {slistSW[3].VertScrollBar} Button
click     {slistSW[3].VertScrollBar} Button
click     {slistSW[3].VertScrollBar} Button
click     {/timelist} Button1 "/Sat 19.30"
click     {/timelist} Button1 "/Sat 19.30"
click     {/timelist} Button1 "/Sat 19.30"
click     {/namefield} Button1 :0 2281
text      {/namefield} "Tullio" 0
press     {/phonefield} Button1 :0 2500
release   {/phonefield} Btn1Down+Buttc
text      {/phonefield} "543232" 2205
click     {/Book} Button1 @43,17 2428
click     {slistSW[3].VertScrollBar} Button
click     {slistSW[3].VertScrollBar} Button
```

```
MovieGuide – Root task
 Session
 Quit
   Cancel reservation
   Make reservation
   Search
   See database part
     Select/insert data
     Book
     CancelBook
     Insert criteria
     Send query
     Choose viewstyle
     Choose a movie
     More info
     Done
       [Where to sit]
       SelData
```

```
Log:  click     {/Book} Button1 @43,17 24
Task: Book
Current logfile: tulio.tcl
```

Fig. 12. The preparation part of USINE.



```
The result from the evaluation:     Current logfile: tulio.tcl

Currentlog: click      {/More info\.\.\.} Button1  @43,19 1903
-----> Preconderror: the precond "Choose a movie" not completed for the task: "More info"

Currentlog: click      {canvas[12].slistSW[2].slist} Button1 "/Stealing Beauty" 9765
Task "Choose a movie" with precondition "Choose viewstyle" completed
Available tasks:
Choose a movie, Choose viewstyle, Choose director, Choose category, Premiers, All movies, More info, Do

Currentlog: click      {/More info\.\.\.} Button1  @37,14 1228
Task "More info" with precondition "Choose a movie" completed
Available tasks:
Vote, Watch trailer, Read reviews, Read about, Close

Currentlog: click      {/About} Button1  @25,21 16749
Task "Read about" without preconditions completed
Available tasks:
Read about, Vote, Watch trailer, Read reviews, Close

Currentlog: click      {/About} Button1  @36,18 4130
Task "Read about" without preconditions completed
Available tasks:
Read about, Vote, Watch trailer, Read reviews, Close

Result file: evalres.eva
The results in temporal order
```

Fig. 13. The result part of the evaluation.

```
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 accomplish 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}
```

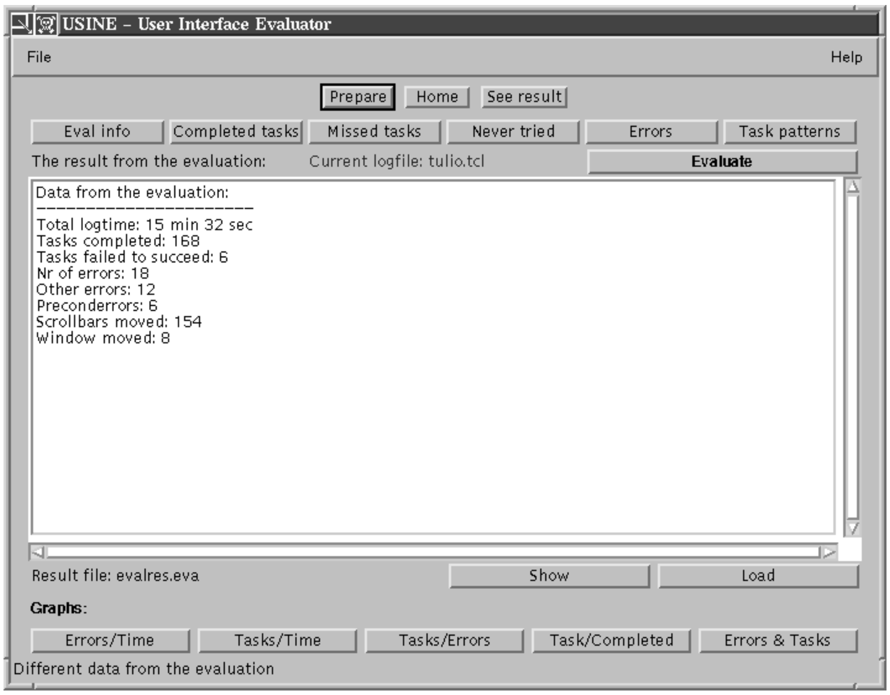
The upper set of buttons provides different ways for the user to present the results:

- The *Eval info* button activates the display of various data from the evaluation such as the number of accomplished and missed tasks, the number of errors and the time the test lasted (see Fig. 14).
- The *Completed tasks* button activates the display of the accomplished tasks and how many times they are performed. The frequency of the tasks is 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 [22].
- The *Missed tasks* button activates the display of the tasks the user tried to perform but failed because their preconditions were not satisfied, and how many times each task failed.
- The *Never tried* button activates the display of the tasks the user never tried to perform.
- The *Errors* button activates the display of all the errors divided into precondition errors and others.
- The *Task patterns* button activates the display of the task patterns found (specific sequences of tasks) among the accomplished tasks (see Fig. 15). The presentation shows first the frequency and next the pattern, and orders them by frequency.

The *Show* button (underneath the result window) activates the display of 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.

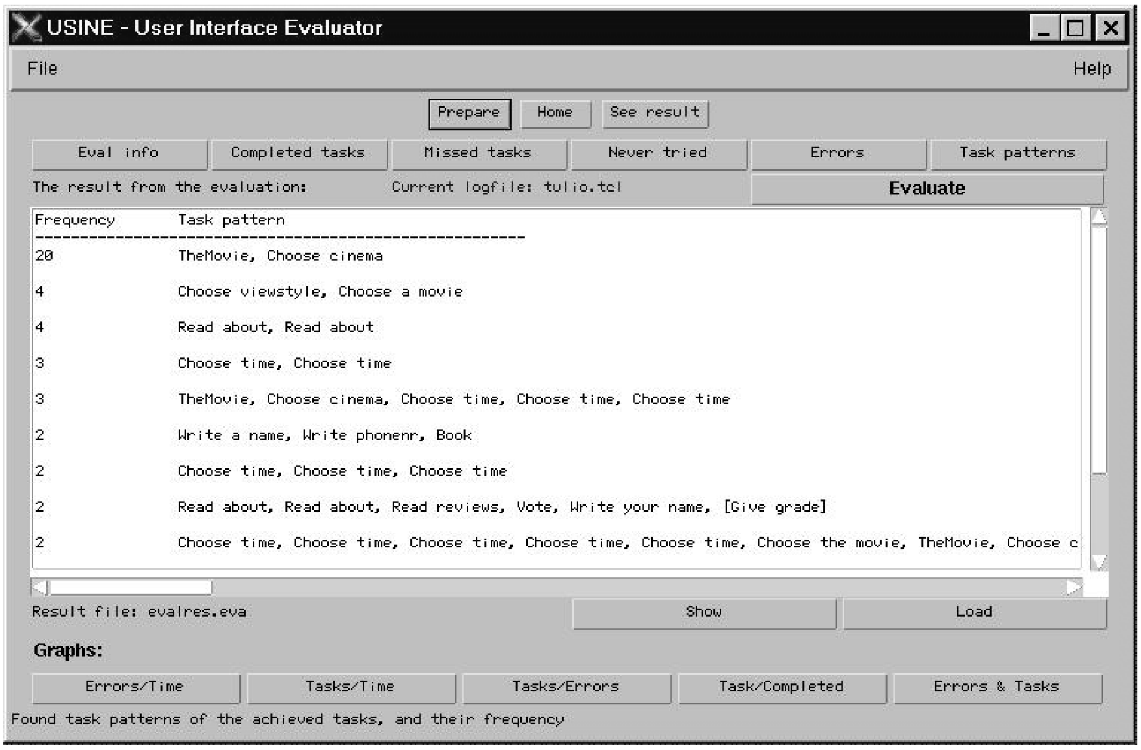
Finally, the lower set of buttons, provides different graphs showing the data from the evaluation in different manners:

- The *Errors/Time* button shows a graph with the number of errors on the y-scale and the time on the x-scale (see Fig. 16).
- The *Tasks/Time* button shows a chart graph with the tasks on the x-scale and how long they took to perform on the y-scale (see Fig. 17).
- The *Tasks/Errors* button shows a chart graph containing the number of precondition errors associated with each task.
- The *Tasks/Completed* button shows a chart graph containing the number of times the tasks were performed.
- The *Errors & Tasks* button shows one pie chart containing the different types of errors and their percentage, and another containing the number of the tasks accomplished, the tasks missed and those never attempted (see Fig. 18).



```
Data from the evaluation:
-----------------------
Total logtime: 15 min 32 sec
Tasks completed: 168
Tasks failed to succeed: 6
Nr of errors: 18
Other errors: 12
Preconderrors: 6
Scrollbars moved: 154
Window moved: 8
```

Fig. 14. The data from an evaluation.



The result from the evaluation: Current logfile: tulio.tcl

```
Frequency       Task pattern
-------------------------------------------------------
20              TheMovie, Choose cinema
4               Choose viewstyle, Choose a movie
4               Read about, Read about
3               Choose time, Choose time
3               TheMovie, Choose cinema, Choose time, Choose time, Choose time
2               Write a name, Write phonenr, Book
2               Choose time, Choose time, Choose time
2               Read about, Read about, Read reviews, Vote, Write your name, [Give grade]
2               Choose time, Choose time, Choose time, Choose time, Choose time, Choose the movie, TheMovie, Choose c
```

Result file: evalres.eva

Found task patterns of the achieved tasks, and their frequency

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

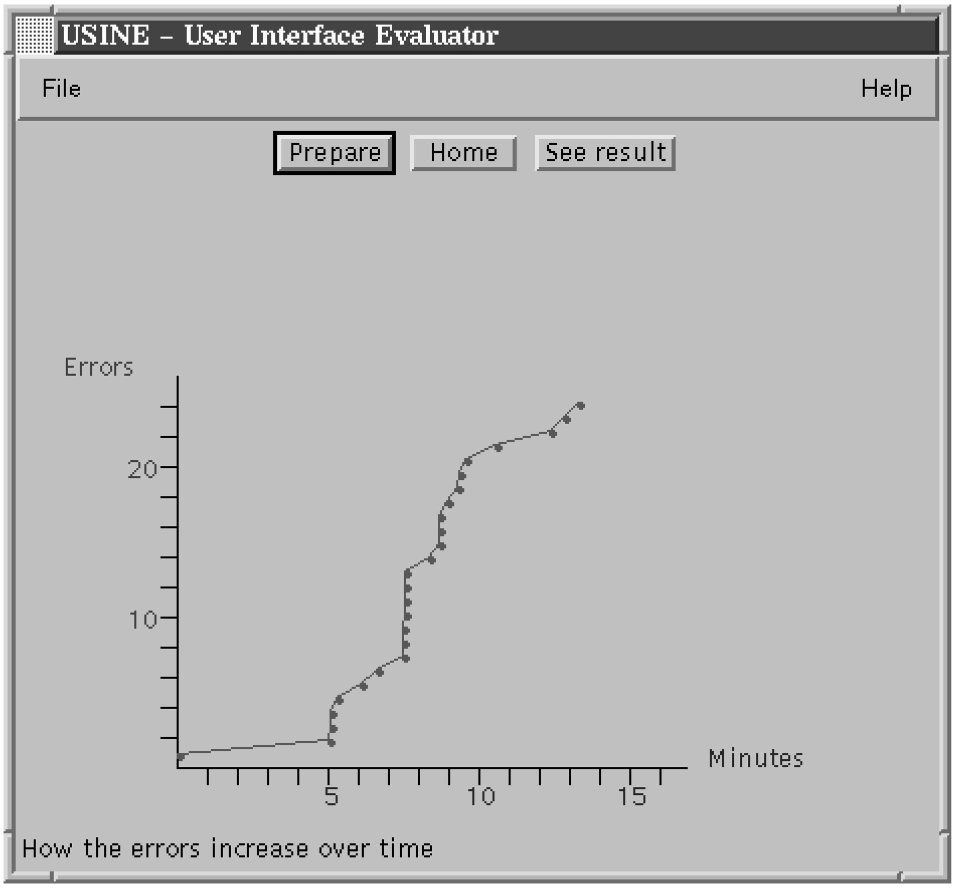
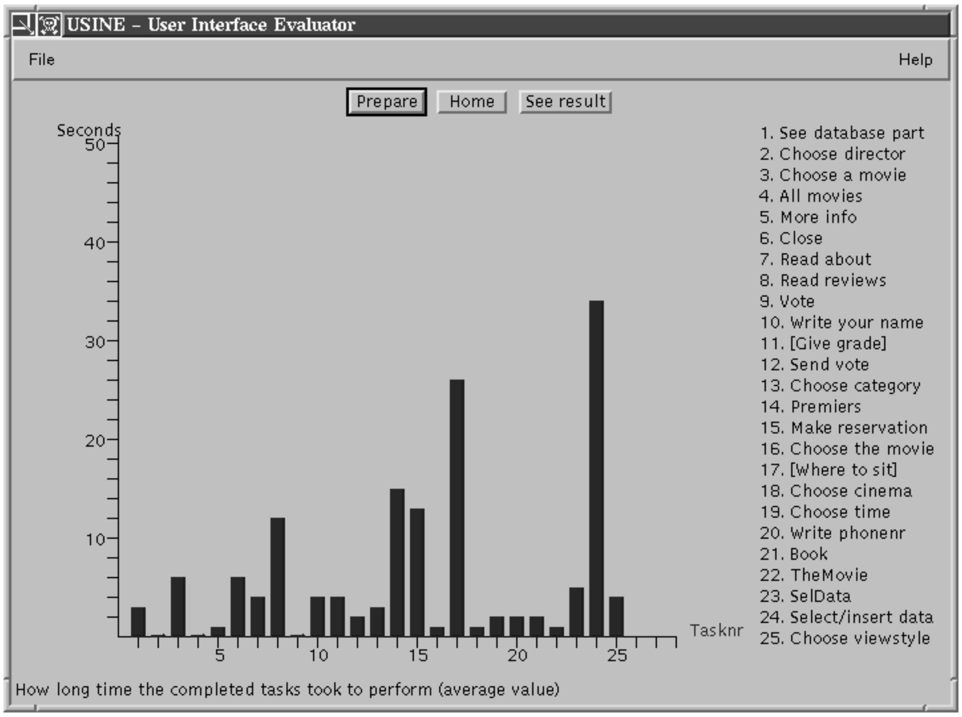


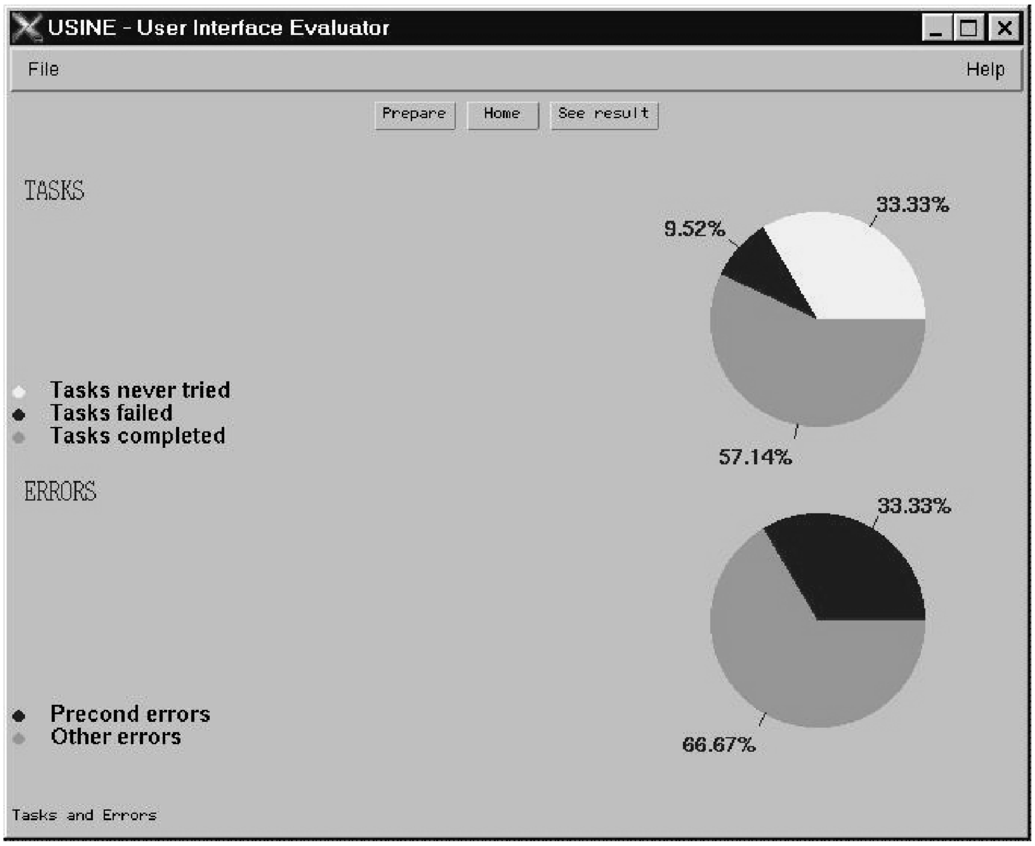
Fig. 16. When the errors occurred during the user test.



1. See database part
2. Choose director
3. Choose a movie
4. All movies
5. More info
6. Close
7. Read about
8. Read reviews
9. Vote
10. Write your name
11. [Give grade]
12. Send vote
13. Choose category
14. Premiers
15. Make reservation
16. Choose the movie
17. [Where to sit]
18. Choose cinema
19. Choose time
20. Write phonenr
21. Book
22. TheMovie
23. SelData
24. Select/insert data
25. Choose viewstyle

How long time the completed tasks took to perform (average value)

Fig. 17. How long each task took to perform.



○ Tasks never tried
● Tasks failed
● Tasks completed

● Precond errors
● Other errors

Tasks and Errors

Fig. 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 *MovieGuide* application which is a Java application. In this example we used it in the X Window System environment (UNIX).

6.1 The User Interface of the Application

The MovieGuide application (see Fig. 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. It is also possible to cancel a reservation and to search for movies in the database.

The database part gives the opportunity to search for 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 people have voted. The reservation part (see Fig. 20) enables users to choose a movie from a list and then make a reservation at a certain cinema, at a certain time and for a number of people. It is also possible to decide where you want to sit in the cinema—at the front, the middle or the back.

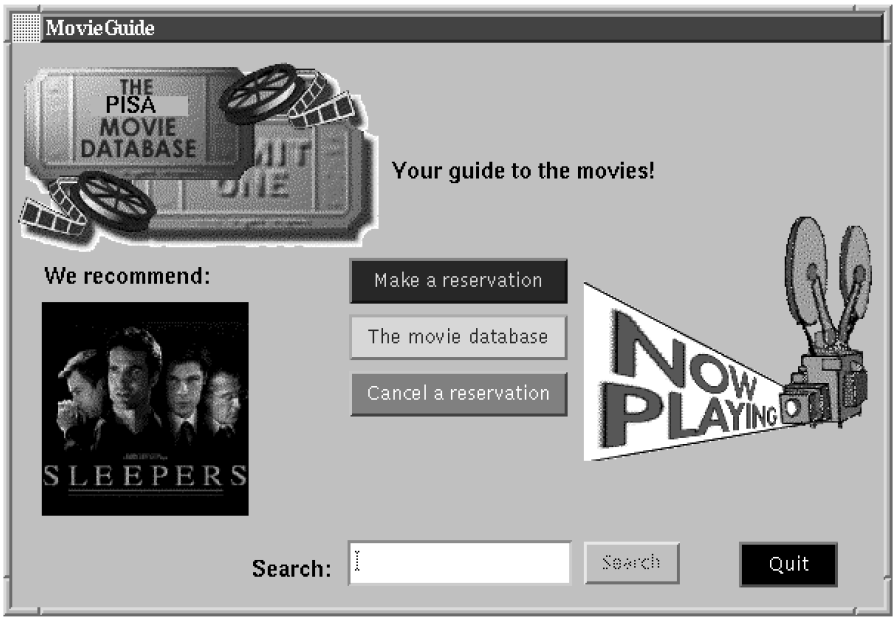
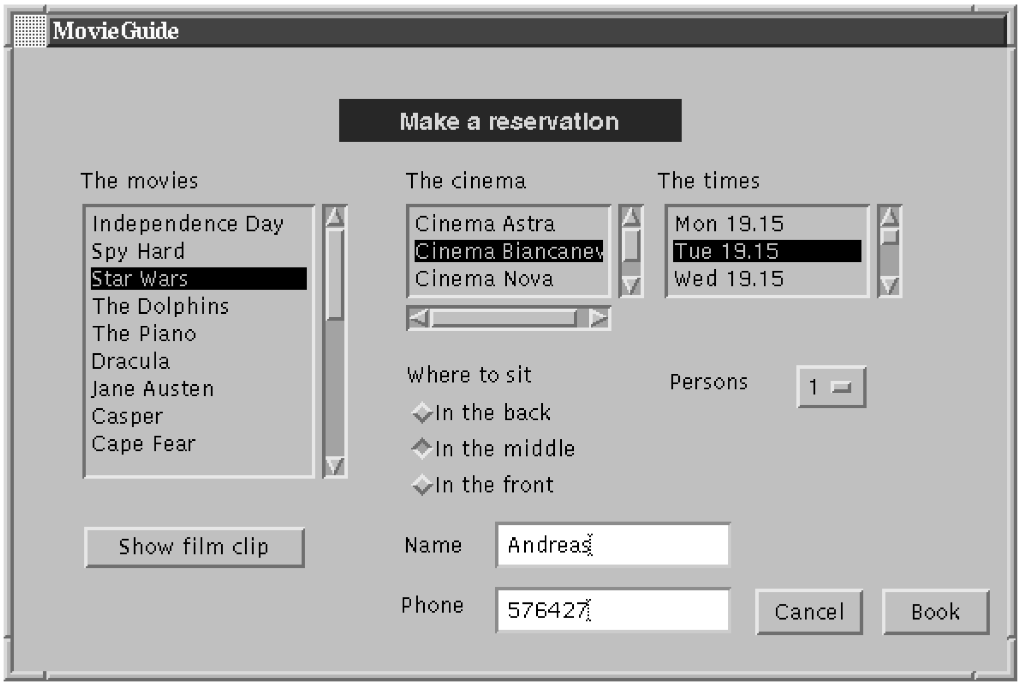


Fig. 19. The MovieGuide, main window.



Independence Day
Spy Hard
Star Wars
The Dolphins
The Piano
Dracula
Jane Austen
Casper
Cape Fear

Cinema Astra
Cinema Biancanev
Cinema Nova

Mon 19.15
Tue 19.15
Wed 19.15

Persons 1

Name Andreas

Phone 576427

Fig. 20. The reservation part of MovieGuide.

6.2 The Task Model of the Application

The task model of the application was made with the editor for specifications in ConcurTaskTrees, the notation described in Section 3. The task model was made *according to* the existing user interface of the MovieGuide application. The reason for this is because we want to evaluate an existing interface, not develop a new one. The task tree was made so deep that every basic interaction task has a corresponding action in the user interface.

Other characteristics of the task model include the following:

- The *Quit* task disables the whole session of the application.
- The *Session* task is divided into four parts with the choice operator between the parts. The different parts are: *Cancel reservation*, *Making reservation*, *Search* and *See database part*. The effect of the choice operator is that when one of the children of *Session* is performed their brothers are no longer available.
- To be able to perform the *Book* task the user first has to fill in the necessary data. 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).
- After the view style is chosen (*Choose viewstyle* task) it is possible to perform the *Choose a movie* task, enabling the *Requesting Info* optional task
- 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 *Nr of persons* task, has the default value “1”. If the user is satisfied with “1” s/he will not do anything, i.e., no interaction with the interface will occur.

6.2.1 The Precondition Table of the Example

From the task tree we automatically create the precondition table by the method described in Section 4. This gives the following precondition table of the task model (read as, **`y | has x as a precondition`**):

In the precondition table we can see that *Select/insert data* has four preconditions—*Make reservation, Write a name, Write phonenr* and *SelData*. The *Choose viewstyle* task on the other hand, needs only one of the tasks, *Choose director, Choose category, Premiers,* or *All movies* to be accomplished (other than the *See database part* task) because only one of these four tasks can be chosen. *ChooseTime* enables *SelData* because *ChooseTime* is the last subtask of *SelData*: this means that whenever *ChooseTime* is terminated then also the *SelData* task is completed

6.2.2 The Log-Task Table of the Example

The log-task table was created by considering a log file generated by a user session. The purpose of this table is to associate the actions from the log file with the basic tasks in the task model. The abstract tasks were marked “#”. This gives the log-task table in Fig. 22.

Task	Precondition	Task	Precondition
MovieGuide MovieGuide	| Session | Quit	All Movies Choose a movie	| See database part | Choose new style
Session	| Cancel Reservation	Request Info	| Choose a movie | Voting | Watch trailer | Read reviews | Read about
Session	| Making Reservation	Request Info	| Choose a movie | Close
Session	| Search	More Info	| Choose a movie
Session	| Managing database	Done	| Choose a movie
Making Reservation	| Book	SelData	| Choose Time
Making Reservation	| CancelBook	Voting	| More Info | Performing vote
Search	| SendQuery	Voting	| More Info | Cancel vote
Managing database	| Done	Watch Trailer	| More Info
Select/insert data	| Make reservation | SelData | Write a name | Write phonenr	Read Reviews	| More Info
SelData	| Make reservation	Read about	| More Info
Write a name	| Make reservation	Close	| More Info
Write phonenr	| Make reservation	The movie	| Choose the movie
Book	| Select/insert data	Choose cinema	| The movie
CancelBook	| Select/insert data	Choose time	| Choose cinema
Send query	| Insert criteria	Performing vote	| Send vote | Vote
Choose view style	| Choose director | See database part	Cancel vote	| Vote
Choose view style	| Choose category | See database part	Give grade	| Vote
Choose view style	| Premiers | See database part	Comment	| Vote
Choose view style	| All Movies | See database part	Write your name	| Vote
Choose director	| See database part	Send Vote	| Write your name
Choose category	| See database part	See Clip	| Choose the Movie
Premiers	| See database part		

Fig. 21. The precondition table of the example.

```
#                                           |MovieGuide
#                                           |Session
click    {/Quit}                            |Quit
click    {/Cancel a reservation}            |Cancel reservation
#                                           }Making reservation
click    {/Make a reservation}              |Make reservation
#                                           |Search
click    {/The movie database}              |See database part
#                                           |Select/insert data
click    {/Book}                            |Book
click    {/Cancel}                          |Camce;Bppl
text     {/searchfield}                     |Insert criteria
click    {/Send}                            |Send query
#                                           |Choose viewstyle
click    {canvas[12].slistSW[2].slist}      |Choose a movie
#                                           |[Requesting info]
click    {/More info\.\.\.}                 |More info
#                                           |Voting
click    {/Done}                            |done
click    {/In the front}                    |[Where to sit]
#                                           |SelData
text     {/namefield}                       |Write a name
text     {/phonefield}                      |Write phonenr
release  {/4}                               |[Nr or 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
#                                           |The Movie
click    {/cinemalist}                      |Choose cinema
click    {/timelislt}                       |Choose time
click    {/1}                               |[Give grade]
test     {textA}                            |[Comment]
text     {textfield}                        |Write your name
click    {/OK}                              |Send vote
click    {/movielist}                       |Choose the movie
click    {/See film clip}                   |[See clip]
```

Fig. 22. The log-task table of the example.

6.3 The User Test of the Application

The user test of the application was performed by 18 different users. The testers were computer science students aged from 24 to 26 who had never seen the application before and were, therefore, total beginners. The users were given written instructions (see Fig. 23) on 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 a general type like "What do you think you could do"? No further help was given during the test.

The test consisted of three goals the user had to reach which were chosen in such a way to force the user to examine different features of the application. During the user test the logs were recorded using the Replay tool (see Section 5). After the test the users answered three questions concerning their impression of the application:

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?

6.4 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 made. 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 was performed following two different approaches. The first approach was based on observations made during the performance of the 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 immediately after a test was finished.

MovieGuide

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

Please try to reach the three following goals:

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 at the front.

Fig. 23. The instruction for the users.

6.4.1 Evaluation Based on Observations

The authors observed the users during the tests of the application, 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 as to 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.

During the test we did not give any time constraint to the users and users did not receive any help from the observers.

The time users took to fulfill the three goals (indicated in Fig. 23) was different.

- *First goal*: minimum time: 0.18 min, maximum time: 4.02 min, average time: 1.69 min.
- *Second goal*: minimum time: 5.38 min, maximum time: 16.57 min, average time: 8.93 min.
- *Third goal*: minimum time: 2.40 min, maximum time: 7.10 min, average time: 4.65 min.

The most important observations were:

- The users clicked on images because they thought they were buttons.
- One problem was that the lists (containing movies, etc.), when shown for the *first* time, were empty. Thus the user clicked in empty lists because they did not understand how to get the information from the list. For example, to get the list of the times when the movies were shown they first had to choose a cinema.
- Double-clicking on items, where a single click was sufficient.
- The scrollbars in the timelist were too small.
- Users tried to write in the empty list.
- Users forgot to write their telephone number.
- Users did not understand immediately that they had to click in a text field before writing in it.
- Users had problems returning to the main window.
- Users pushed the "More info…" button twice thus getting two windows.

Therefore, an improved version of the application should include improvements on these issues. However, it was expensive to obtain these suggestions because they required an evaluator completely dedicated to observe users while they were working with the application.

6.4.2 The Answers to the Questions

We produced a short questionnaire. The questionnaire was not meant to be exhaustive. We provided a limited set of questions because in this case the questionnaire had a limited purpose: to get the subjective feeling of the users about their experience with the use of the application, to receive directly from the user comments on the main usability problems found and to know whether the users would be willing to use the application during their every day life.

A summary of the answers includes the following:

1) Did you have any problems during the test?
   - 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 cinema list and the time list.
   - The controls of the video presentation.
   - To return to the main window.
   - Not enough feedback when moving the cursor.
   - At the beginning it was difficult because it was all new, but then it was easy.
2) Which part was the most difficult (if any), or the most irritating?
   - Thought the images were buttons.
   - The time list was too small.
   - Many distracting images.
3) Would you use this software (an improved version) in the future? Why, or why not?
   - Yes, probably, it is useful because it allows easy access to a useful service from home and work.

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.

6.5 Evaluation by the Proposed Method

Based on the task model and the user logs we performed an evaluation according to the method described in Section 4. We used the USINE tool (see Section 5) to get evaluation data from the tests. In counting the tasks which were accomplished, failed or never tried the tool counted both high-level tasks and their subtasks (including also basic tasks) which were indicated in the task model and which were involved during the session to reach the goals indicated.

The average results obtained (the averages were calculated on the data of all the 18 user sessions) are:

Accomplished tasks:	24
Failed tasks:	13
Never tried tasks:	5
Other errors:	7
Scrollbars moved:	56
Window resized:	9

We found that all users accomplished the three main goals (they were the same three goals for all the users) described in the instructions (see Fig. 23) although during the sessions they made some errors and thus failed, at least at the first attempt, to perform some tasks or subtasks which were then eventually performed correctly during the same session. This indicates that the user interface was easy to use. The time it took the users to perform the tasks was too long with respect to what is strictly required to perform the actions needed to accomplish the goals indicated. This means that some improvements in the design were needed.

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

We also got information as to how many *times* each task was performed. It showed that the *Choose a movie* and *Choose cinema* tasks were performed most.

The scrollbars were used frequently and the window was resized a few times. This could indicate that these lists (as currently designed) did not make it easy to find the desired information.

6.5.1 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 users failed to perform before the current task.

The reasons for the precondition errors (failed tasks) can guide us as to 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 accomplish and the reason for the error.

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

The above means that the user did not, as required, perform the *Choose a movie* task before the current task, *More info*. When a task with a precondition is completed we will get a result as 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 preconditions were not satisfied when the user failed to accomplish a task. The task (or tasks) associated with the unsatisfied precondition then indicates where improvements should be made to the user interface.

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

- For *Choose Time* it was because the user after having selected a movie had to choose a cinema before it was possible to choose the time. The user tried to choose the time before the cinema had been chosen.

An improvement to the interface would be to enable users to select the time before the cinema so that the user can see where the movie of interest is showing at that time. This would allow users to better perform tasks such as the third in our test (*Independence Day* or *Sleepers* at 19.30) where a movie and a time are predefined but not the cinema.

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

The interface could be provided with more information and help on how to perform the task *More info*. For example there could have been a more explicative label indicating that you must choose a movie before you can view more info.

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

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

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

The improvements in this case could include message boxes occurring 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, one way could be to highlight the fields that must be completed.

This case study shows that it is possible to find various causes for the errors detected. Possible examples are:

- the user interface can impose temporal constraints in performing tasks which are too rigid for the conceptual model of the user of the activities to perform;
- the user interface does not provide enough guidance on how a task should be performed
- a label is not sufficiently explicative.

The reasons for the errors do not say explicitly *how* to improve the interface. However, they show that improvements must be made and indicate which part of the design of the user interface should be improved. The performed improvements must then be decided by the designer, helped by these results.

These errors may be found by observing users. However, our tool-supported method can guarantee a more reliable and precise detection of these errors, especially if the application is complex or things happen fast, and these errors are detected automatically without having an observer spending time on analyzing user interactions.

6.5.2 The Never Tried Tasks

Among the never tried tasks we found the *See a film clip* task located in the reservation part. We noticed that we had an inconsistency between the *Trailer* task 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 tightly connected to the other tasks in this part. That is, it is not a part of the tasks users normally perform when they want to make a reservation.

This is an important result of our method as it was not detected by interviewing and observing users and it is often harder to find something that never happened with respect to something that happened incorrectly.

6.5.3 Other Errors

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

6.5.4 The Temporal Order of the Tasks

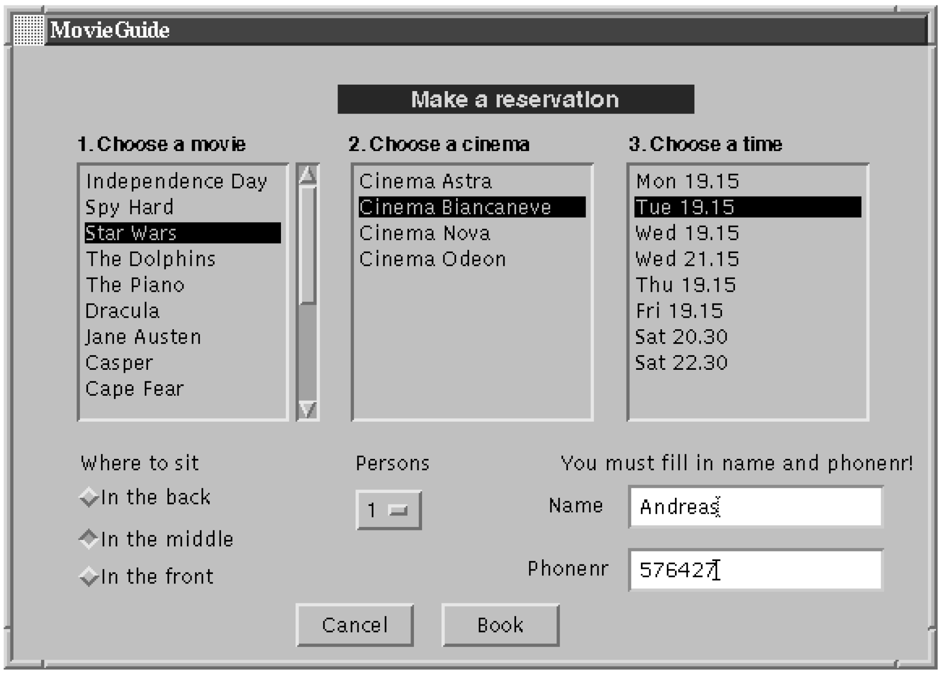
Among the *task patterns* we found that the user often chose a cinema after s/he had chosen a movie. This is an indication that the application after the user has chosen a movie should automatically provide the cinemas that are showing it.

6.6 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 add the possibility to select time first and then the cinema
- To make the difference between buttons and images more distinct
- To insert 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" and "nonclickable" objects
- To make the information visible in all the lists from the beginning
- To make the lists bigger
- To allow double-click on list items.
- To add more labels and messages, providing useful and helpful information
- To take away the possibility of seeing a film clip in the reservation part

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


MovieGuide

Make a reservation

1. Choose a movie: Independence Day, Spy Hard, Star Wars, The Dolphins, The Piano, Dracula, Jane Austen, Casper, Cape Fear

2. Choose a cinema: Cinema Astra, Cinema Biancaneve, Cinema Nova, Cinema Odeon

3. Choose a time: Mon 19.15, Tue 19.15, Wed 19.15, Wed 21.15, Thu 19.15, Fri 19.15, Sat 20.30, Sat 22.30

Where to sit: In the back, In the middle, In the front

Persons: 1

You must fill in name and phonenr!

Name: Andreas

Phonenr: 576427

Fig. 24. The new version of the reservation part, after the evaluation.

6.7 Comparison of the Two Evaluation Approaches and their Results

The two approaches found some similar usability problems with the application. However, the results from the evaluation based on the observations are *supported* and clearly identified by the results from the evaluation made 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. Furthermore, with our tool-supported method we found additional problems which were not discovered by observing and interviewing users, such as identifying tasks which were not performed, thus indicating parts of the user interface difficult to find for the end user or supporting unnecessary tasks.

Through the observations we found that the user clicked in empty lists to get some information from the lists. With our method though, we found the same problem but we also automatically got which task caused the error and which task the user should have performed first to avoid the error. This may be detected by observers but they may have problems finding it if the user interaction evolved fast or when the users' behavior does not make clear what their intentions are. In other words, without the cost of an observer and in a more reliable way we can detect the error, i.e., the actual precondition was not satisfied and have useful information for improving the user interface. 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.

The results from the proposed method can give useful information for deciding how the designer should improve the user interface. We can also give specific information, to say *where* in the interface improvements should be made. This is due to the precondition errors, pointing out the tasks that caused the user problems. However, the final decision of whether improvements are to be made and how they should be done, is still up to the designer.

Our method gives specific information, 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 [4]) of the application could be, e.g., two precond-errors, two other errors, and all tasks achieved. With these goals in mind we can improve and redesign 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.

Another advantage of our approach is that it provides consistent information with respect to all the sessions analysed whereas the results of observation-based approaches may vary from one observer to another and one session to another.

6.8 Evaluation of our Approach

One reasonable question is whether the benefits of our new approach justify the extra time and effort involved.

The additional effort required only regards the design of the task model for the application considered and to make the log-task table (all the other tables in the preparation part are automatically created completely).

We think that the task model is a useful exercise for user interface designers to develop an understanding of the application considered and it should be done even if our approach for user interface evaluation was not used. Indeed, at this time the most common use of the task model is in the design phase and not in the evaluation phase. It is used to discuss design solutions among the different people involved (designers, developers, managers, end users and so on). Furthermore, the task model can also be useful to support the software development phase. We have developed a method of using it to drive systematically the development of the user interface [17].

The mapping between user actions in the log file and tasks should be done only once for an application and then it is valid for analyzing all the user tests of that application. Thus it is a limited additional effort.

On the other hand, it is possible to have more reliable evaluations of the application as no action to be analysed can be omitted, we can detect automatically when they occur, and the approach is also valid for large applications. Additional results such as tasks never tried are provided. When our approach gives similar results to those obtained by observing users, it has some advantages because in our case we can run various user tests in parallel in the users' workplaces without having to move them to a usability lab or to move observers to their workplaces. Then we can let the automatic tool evaluate the results whereas in the other case we need an observer to follow sequentially each user and analyse manually his/her behavior with great effort in terms of time from both the user and the observer and less reliability of the results especially when user interactions evolve rapidly. If multiple observers are used then there is the problem of consistency among the results of their observations.

One potential concern with our method is that the evaluators could be missing some important information that would only be gained by actually observing users and hearing their comments. We thus propose its use complemented with user interviews to gather direct comments. The questions would be limited to the part of the interface that our tool indicates the user found difficult to use.

Overall, we believe that our approach is worth using especially if the application considered has many dynamic dialogues and a substantial complexity. It complements and strongly reduces the need for evaluation done by observing and interviewing users which can be limited to an additional analysis of the parts of the user interface which our method finds problematic.

7 Conclusions and Future Work

We have presented a method based on the use of a task model that allows the designer to specify a rich set of temporal relationships among tasks and the association between user actions and basic tasks. This makes it possible to find the tasks causing the user problems, and therefore to find the parts of the user interface that should be improved.

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

Another advantage of our method is that its results give the evaluator the 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 two minutes," or "the user may only make two errors in this task." If the evaluator sets 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 of evaluating a user interface that finds the tasks and the errors performed during a user test, and the reasons for most of the errors. To this end, we have found a method of getting the preconditions from the task model and the available tasks from the current state of the user session. The latter is not only useful for evaluation purposes but also for supplying built-in task oriented help.

With respect to user testing our method can give the evaluator more specified and concrete results as the tasks and errors are performed. It does not give explicit recommendations for the evaluator on how to improve the user interface. However, it can give support when deciding which improvements should be made and where in the interface they should be made. This is possible due to the precondition errors which point out those tasks that caused user problems. We propose its use complemented with users' interviews limited to the part of the user interface which is found difficult for end-users by our tool.

The definition of usability includes relevance, efficiency, attitude, learnability and safety. The proposed evaluation method includes the possibility of measuring efficiency and to a certain degree learnability and relevance. Learnability can, for example, be measured by the time completion rate and the number of errors. Relevance can be measured by examining whether the user succeeded in completing the desired tasks. However, we have not covered attitude and safety. Future work could thus be directed to investigate other aspects of usability.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their comments which were very useful for improving the content and the presentation of the paper.

REFERENCES

[1] N. Bevan, "Measuring Usability as Quality of Use," *Software Quality J.* vol. 4, pp. 115–130. Chapman & Hall, 1995.

[2] S.K. Card, T.P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*. Hillsdale, N.J.: Lawrence Erlbaum Assoc., 1983.

[3] D.Diaper, *Task Analysis for Human-Computer Interaction*. Chichester: Ellis Horwood, 1989.

[4] J.S. Dumas and J.C. Redish, *A Practical Guide to Usability Testing*. Norwood, N.J.: Ablex, 1994.

[5] M. Good, T.M. Spine, J. Whitside, and P. George, "User-Derived Impact Analysis as a Tool for Usability Engineering," *Proc. Conf. Human Factors in Computing Systems, CHI'86*, M. Mantei and P. Oberton, eds., pp. 241–246. New York: ACM Press, 1986.

[6] D. Hix and H.R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process*. New York: John Wiley & Sons, 1993.

[7] ISO Information Processing Systems—Open Systems Interconnection—LOTOS—*A Formal Description Based on Temporal Ordering of Observational Behavior,* ISO / IS 8807. ISO Central Secretariat, 1988.

[8] R. Jeffries, J.R. Miller, C. Wharton, and K.M. Uyeda, "User Interface Evaluation in the Real World: A Comparison of Four Techniques," *Proc. Conf. Human Factors in Computing Systems, CHI'91,* pp. 119–124, ACM Press, 1991.

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

[10] D. Kieras, W. Scott, and D. Meyer, "Predictive Engineering Models Based on the EPIC Architecture for a Multimodal High-Performance Human-Computer Interaction Task," *ACM Trans. Computer-Human Interaction*, vol. 4, no. 3, pp. 230–275, Sept. 1997.

[11] C. Lewis, "Using the 'Thinking Aloud' Method in Cognitive Interface Design," Research Report RC9265, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1982.

[12] C. Lewis and J. Rieman, *Task-Centered User Interface Design: A Practical Introduction*, a shareware book published on the web by the authors: www2.umassd.edu/Coursepp./HCI/hcireadings/TextVersion/index.htm. Also available at ftp.cs.colorado.edu/pub/cs/distribs/clewis/HCI-Design-Book, 1993.

[13] M. Macleod, R. Bowden, and N. Bevan, "The MUSiC Performance Measurement Method," *HCI'96,* Tutorial 14, *Measuring Usability—MUSiC Methods*, N. Bevan, London: The British HCI Group, 1994.

[14] J. Nielsen, *Usability Engineering.* Boston: Academic Press, 1993.

[15] S. Pangoli and F. Paternò, "Automatic Generation of Task-Oriented Help," *Proc. ACM Symp. User Interfaces Software and Technology*., pp. 181–187, Pittsburgh: ACM Press, 1995.

[16] F. Paternò, C. Mancini, and S. Meniconi, "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models," *Proc. Interact'97,* pp. 362–369, Sydney: Chapman & Hall, 1997.

[17] F. Paternò, C. Mancini, and S. Meniconi, "Engineering Task Models," *IEEE Conf. Eng. Complex Systems*, pp. 69–76, Como, IEEE CS Press, Sept. 1997.

[18] F. Paternò, M.S. Sciacchitano, and J. Löwgren, "A User Interface Evaluation Mapping Physical User Actions to Task-Driven Formal Specifications," *Design, Specification and Verification of Interactive Systems'95. Proc. Eurographics Workshop*, P. Palanque and R. Bastide, eds., pp. 35–53, Toulouse: Springer-Verlag, 1995.

[19] P.G. Polson, C. Lewis, J. Rieman, and C. Wharton, "Cognitive Walkthroughs: A Method for Theory-Based Evaluation of User Interfaces," *Int'l J. Man-Machine Studies*, vol. 36, pp. 741–773, 1992.

[20] *Human-Computer Interaction,* J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey, eds. Workingham, England: Addison-Wesley, 1994.

[21] A. Sears, "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout," *IEEE Trans. Software Eng*., vol. 17, no. 7, pp. 707–719, July 1993.

[22] A. Sears, "AIDE: A Step Toward Metric-Based Interface Development Tools," *Proc. UIST'95*, pp. 101–110. New York: ACM Press, 1995.

[23] C. Wharton, J. Rieman, C. Lewis, and P. Polson, "The Cognitive Walkthrough: A Practitioner's Guide," *Usability Inspection Methods,* J. Nielsen and R.L. Mack, eds. New York: John Wiley & Sons, 1994.

[24] S. Wilson, P. Johnson, C. Kelly, J. Cunningham, and P. Markopoulos, "Beyond Hacking: A Model-Based Approach to User Interface Design," *Proc. HCI'93.* J.L. Alty et al., eds., *People and Computers VIII, Proc. Conf., HCI'93 Loughborough, UK,* Cambridge: CUP, 1993.

Andreas Lecerof received his MS degree in computer science from Linköping University, Sweden, in 1997. He wrote his master's thesis while at CNUCE-CNR, Pisa, Italy, in 1996. He is currently working as a software engineering consultant at Cap Gemini, Sweden. His research interests include usability engineering, user interface design, and evaluation.

Fabio Paternò received the Laurea degree in computer science from the University of Pisa, Italy, and the PhD degree in computer science from the University of York, UK. Since 1986, he has been a researcher at CNUCE-CNR, Pisa, where he is head of the HCI group. He has worked on various national and international projects on user interfaces-related topics. He is the coordinator of the Modeling Evaluating and Formalizing Interactive Systems Using Tasks and Interaction Objects (MEFISTO) Long Term Esprit European Project. He has developed the ConcurTaskTrees notation for specifying task models, which has been used in many industries and universities, and related methods for supporting the design of user interfaces. His current research interests include methods and tools for user interface design and usability evaluation, formal methods for interactive systems, and design of user interfaces for safety critical interactive systems. Dr. Paternò was chair of the first International Workshop on Design, Specification, and Verification of Interactive Systems. He is co-editor of the book on *Formal Methods in Human-Computer Interaction*. He has been a member of the program committee of the main international HCI conferences. He is a member of the IFIP Technical Committee 13 on Human Computer Interaction.