Human–Computer Interaction — INTERACT'99
Angela Sasse and Chris Johnson (Editors)
Published by IOS Press, © IFIP TC.13, 1999

Model-aided Remote Usability Evaluation

Fabio Paternò & Giulio Ballardin

CNUCE-CNR, Via S. Maria 36, Pisa, Italy.

{f.paterno,g.ballardin}@cnuce.cnr.it

Abstract: In this paper we present a method for providing remote usability evaluation with the support of powerful task models able to describe concurrent and interactive activities. The method is tool-supported and it is able to analyse the logs of user actions performed during work sessions by using information contained in such task models. The overall goal is to provide useful information to usability evaluators with limited effort.

Keywords: usability engineering, remote evaluation, task models, tool-supported evaluation, formal methods for HCI.

1 Introduction

Usability engineering (Nielsen, 1993) concerns the development of systematic methods to support usability evaluation. Various types of approaches have been proposed for this purpose.

Model-based approaches to usability evaluation use some models, usually task or user models, to support this evaluation. They often aim (John & Kieras, 1996) to produce *quantitative predictions* of how well users will be able to perform tasks with a proposed design. 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) to find the usability problems of the interface. While model-based evaluation is useful to highlight relevant aspects in the evaluation, it can be limiting to not consider empirical information because the possible predictions in some cases can be denied by the real user behaviour. Thus, it is important to find methods that allow designers to apply meaningful models to some empirical information. An attempt in this direction is USAGE (Byrne et al., 1994) that provides a tool supporting a method where the user actions required to execute an application action in UIDE are analysed by the NGOMSL approach. However this information is still limited with respect to that contained in the logs of the user actions performed during work sessions by users.

In *empirical testing* the behaviour of real users is considered. It can be very expensive and it can have some limitations too. It requires long observations of users' behaviour. Often these observations are supported by video that can be annotated by some tool. Even observing video describing user behaviour, either in work places or in usability laboratory, can take a lot of time to designers (a complete analysis can take more than five times the duration of the video) and some relevant aspects can still be missed.

In *inspection-based techniques* to usability evaluation designers analyse a user interface or its description. Several of these techniques, such as heuristic evaluation, cognitive walk-through, and software guidelines, have been found useful but limited because dependent on the ability of the evaluator or requiring multiple evaluators, or missing some relevant problems (Jeffries et al., 1991).

In order to overcome some of the limitations considered we developed a method (Lecerof & Paternò, 1998) based on the possibility to use task models for analysing empirical data. We present in this paper an extension of this method (RemUSINE, **Rem**ote **US**er **IN**terface **E**valuator) to exploit its possibilities for supporting remote evaluation. The new method is more efficient and provides a richer set of results such as results related to a group of users sessions rather than just one.

2 The RemUSINE Approach for Remote Evaluation

In the last years there has been an increasing interest in remote usability evaluation (Hartson et al., 1996). It has been defined as usability evaluation where evaluators are separated in time and/or space from users.

This approach has been introduced for many reasons:

- The increasing availability and improvement of network connections.
- The cost and the rigidity of traditional laboratory-based usability evaluation.
- The need to decrease costs of usability evaluation to make it more affordable.

There are various approaches to remote usability evaluation. One interesting approach is automated data collection for remote evaluation, where tools are used to collect and return a journal or log of data containing indication of the interactions performed by the user. These data are analysed later on, for example using pattern recognition techniques, however usually the results obtained are rather limited for the evaluation of an interactive application.

We think that task models can provide additional support for analysing such data. However to this end it is necessary that task models are powerful, non-prescriptive, and flexible. This means they should be able to describe concurrent activities that can interrupt each other dynamically. To support this analysis, task models should be refined to indicate precisely how tasks should be performed following the design of the application considered.

If we consider current approaches, briefly summarised in the introduction, we can notice a lack of methods that are able:

- To support the evaluation of many users without requiring a heavy involvement of designers.
- To support the evaluation gathering information on the users' behaviour at their work place without using expensive equipment.
- To apply powerful and flexible task models in the evaluation of logs of user events, thus linking model-based and empirical evaluations. Current automatic tools, such as ErgoLight, that support usability evaluation by task models, use simple notations to specify such models, thus still requiring a strong effort from the evaluator.

These three relevant results are obtained by our RemUSINE method following an approach that is summarised in Figure 1 where ovals represent data files and rectangles programs.

We have multiple instances of one application that can be used by many users located in different places. Then we have the RemUSINE tool with the following input:

- *The log files with the user interactions*, by the support of a logging tool it is possible to automatically generate a file storing all the events performed by a user during a work session. One or more of these files have an additional use that is the creation of the log-task table.
- *The log-task association table;* the purpose of this table is to create an association between the physical events, that can be generated by the user while interacting with the application considered, and the basic interaction tasks (the tasks that cannot be further decomposed in the task model and require only one action to be performed). This association is a key point in our method because through it we can use the task model to analyse the user behaviour.
- *The task model*, it is specified using the ConcurTaskTrees notation (Paternò, 1999) by using an editor publicly available at http: //giove.cnuce.cnr.it/ctte.html.

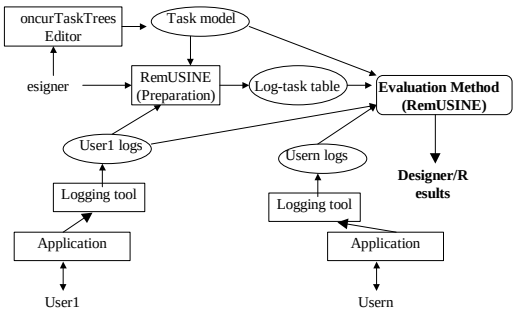

Figure 1: The architecture of our approach.

In our method we can distinguish three phases:

- *The preparation part*, that is mainly the development of the task model and the association between physical user-generated events, extracted by log files, and the basic interaction tasks of the task model;
- *The execution of the evaluation tool*, during which the tool first processes the related precondition for each task (if any) from the temporal relationships defined in the task model, and then, using also this information, it elaborates its results: the errors performed, task patterns, duration of the performance of the tasks and so on;

- *The analysis of the results of the tool*, in this phase the designer can provide suggestions to improve the user interface by using the information generated by the RemUSINE tool.

The user interactions are captured by logging tools that are able to get this information without disturbing users during their work. These logs are useful to identify user errors such as attempts to activate an interaction that was not allowed because some precondition was not satisfied or selections of elements of the user interface that were not selectable. An action is an error if it is not useful to support the current task.

One problem is how to identify automatically the tasks that the user intends to perform. To this end the knowledge of the user actions can be useful because, for example, if the user tries to print a file and s/he does not specify the name of the file to print, it is possible to understand what the current user intention is (printing a file) by detecting the related action, for example the selection of a Print button. Besides, a similar precondition error highlights that there is a problem with the user interface, as it probably does not highlight sufficiently the need to specify the name of the file to print.

3 ConcurTaskTrees Task Models

ConcurTaskTrees is a powerful and flexible notation to specify task models. It aims to overcome limitations of previous approaches such as GOMS (Card et al., 1983) that considers only sequential tasks (rather unrealistic in modern user interfaces where various interactions can be concurrently activated by the user) or UAN (Hartson & Gray, 1992), which made an important contribution by providing concurrent operators, but it entails specifying a lot of details (such as highlighting buttons when the cursor is over them). Such details expressed in UAN textual syntax and its lack of tool support, make the specifications difficult to interpret, especially in real, large-size case studies.

ConcurTaskTrees is powerful and flexible because it allows designers to specify concurrent tasks, tasks that dynamically disable other tasks, iterative and recursive tasks and so on.

By using different icons it is possible to indicate the allocation of the performance of the task (to the user or application, or to their interaction).

In Figure 2 you can find an example of ConcurTaskTrees specification. It concerns a museum application. The related user interface is shown in Figure 3. In this specification we did not consider user tasks (tasks associated with internal cognitive actions such as recognizing a visual element in the interface or deciding how to carry out a set of tasks) because we want to focus on interaction tasks that can be associated with the user actions stored in the log files and application tasks that are tightly related to them.

Figure 2: An example task model.

Figure 3: The user interface considered.

At the first level there is a distinction between the part to specify the request (*SpecifyRequest* task) and that for sending the request (*SendbyButton* task) that can disable ([> operator) the first one. *SpecifyRequest* has iterative children tasks (iterative tasks are indicated by the * symbol) as they can be performed multiple times. These tasks support specifying and clearing the request and they are concurrent communicating tasks (indicated by the |[]| operator) disabled by the sending the request (*SendbyButton* task).

More specifically, specifying a request can be performed either by mouse selection (*SpecCategory*

task) or by typing (*SpecifybyTyping* task). They are communicating because a specification done in one way can override a specification done in the other way beforehand and vice versa. The specification by mouse requires first to select a category (*SelCategory* task), next the application presents the list of related values (*PresList* task) and then the user can select a value (*SelValue* task) and the application shows it in the related field (*ShowValue* task). These are sequential activities with information passing ([] >> operator). Selecting a category is decomposed into the choice ([] operator) among different tasks, each one associated with a specific category of request (by sculptor, by type of work, by historical period, by work name).

Once the list of values is presented the user can activate another request for another field by selecting another category. Specifying by typing requires first to select the field of interest (*SelField* task) and next to type the value (*TypeValue* task).

The cloud icon is used to indicate abstract tasks, i.e. tasks that have subtasks whose performance is allocated differently (either user, application, or interaction tasks).

It is important to remember that for the RemUSINE analysis it is required that the task model is refined so as each possible user action, supported by the user interface considered, can be associated with one interaction basic task (that are leaves in the task tree).

4 The Preparation Part

There are various tools available to collect automatically data on the user-generated events during an interactive session, for example, JavaStar (http://www.sun.com/suntest/JavaStar/JavaStar.html) or QCReplay (http://www.centerline.com/productline/qcreplay/qcreplay.html).

They are able to provide files that contain indication of the events occurred and when they occurred. The events considered are mouse click, text input, mouse movements, and similar. When interaction techniques such as menu, pull-down menu are selected they are able also to indicate what menu element was selected. The resulting files are editable text files.

Similarly, using the ConcurTaskTrees editor it is possible to save the task model specification in a file for further modifications and analysis.

In RemUSINE there is a part of the tool that is dedicated to the preparation part whose main purpose is to create the association between logs of user events and the basic tasks in the task model. Association that will be used to analyse the user interactions with the support of the task model.

Figure 4: The tool supporting the preparation phase.

In the preparation part (as you can see in Figure 4) the evaluator can load one log file and one file with the task model. The lists of elements contained in the two files appear in two different, parallel columns. In the task-related part only the basic interaction tasks appear, as they are the only elements that can be associated with logged events. While in the list of basic tasks each of them is indicated only once, the number of times that one specific event can appear on the related list depends on the session considered and the number of times the user performed it during such a session. The application designer then has to select one physical event on one side and the corresponding basic task on the other side, and then add it to the table containing all the associations by means of the related button. Once a basic task has been associated with the related event it will disappear from the list of tasks that thus indicates only tasks that still need to be associated with the related event by the designer. The associations performed can be displayed by the *Associations* pull-down menu. In case of mistakes the designer can remove elements from this list by the *Remove* button. The associations can be saved in a file and loaded later on for further expansions or for the evaluation phase.

This association can be made only once to evaluate as many user application sessions as desired because it contains the information required by the evaluation tool. Indeed, each session is associated with one event trace whose elements belong to the set of input events supported by the user interface in question. Thus, once the association has been created, only the log file to be analysed needs to be changed in order to evaluate each session.

5 The RemUSINE Elaboration

In the evaluation phase the RemUSINE tool scans the elements of the log considered. For each element it identifies the corresponding basic task by the log-task association table. It has to check whether such a task exists, if not it means that the user performed an error of the type selection of an item that was not selectable. If it exists then it has to check whether it could be performed when it occurred. This is obtained by checking whether the task had preconditions (tasks that have to be performed before the one considered) and, in the event they exist, if they were satisfied. If yes then the task can be considered executed correctly otherwise a precondition error is annotated giving an indication of what precondition was not satisfied. Then the next user action in the log file is considered and the same elaboration is applied.

When a task is performed correctly the internal data structure of the tool are updated to maintain updated the context that will be used in the evaluation of the next user actions.

An example of task with precondition in Figure 2 is the task related to typing a value in one field of the request form. If the user has not selected such a field this task cannot be performed and this has a consequence also on higher level tasks such as formulating the request because they too cannot be performed correctly.

In the tool we have incorporated the implementation of an algorithm that takes a ConcurTaskTrees specification and it is able to provide the preconditions for all the tasks at all the levels. Such preconditions indicate the tasks that have to be performed in order to complete the accomplishment of the task considered.

The method for finding the preconditions and creating the precondition table searches through a pre-order traversal of the task tree. For every non-optional task we check if its *left brother is an enabling task* (a task on the left of the $>>$ operator). If so we know that the left brother is the precondition of the current task and add this (the task and its precondition) to the result.

If the current task is a high level task then the left enabling brother is a precondition also for the children of such a task that are available at the beginning of its performance. These children are those which are on the left of the leftmost enabling operator. An example is the case of *SpecValue* in Figure 2 with the left enabling brother *PresList*. We will first put *PresList* as the precondition of *SpecValue*. Then we find also that *PresList* is a precondition for *SelValue*. If there is no enabling operator it means that all the children are available at the beginning.

In any case when we consider a non-basic task (a task that is not a leaf in the task tree) we have to search for its preconditions which are among its children. While searching the method collects the results. For example, if we apply the method to the example in Figure 2 we will find that *SendbyButton* is precondition of *AccessVirtualMuseum* because only when this task terminates the parent task will be considered completed as the brother task (*SpecifyRequest*) has iterative children that never terminate unless they are interrupted by the disabling task. At the next levels of the task tree we can find that *SpecifybyTyping* has *TypeValue* as precondition which has *SelField* as precondition. The algorithm can continue similarly until the entire tree has been considered.

Another element that we have to take into account is that sometimes the performance of one task has the effect of undo the effects of another task thus making unverified a precondition that beforehand was true. For example we can have a deselect task that makes false a precondition requiring a certain selection. The tool considers this possibility too when it keeps updated the state of the task model for evaluation purposes.

6 A Small Example of Log Analysis

We can consider a small example of log taken using JavaStar applied to the museum application in Figure 3 to better understand how our method works once it has calculated the preconditions for all the tasks. In the log we have removed the information useless for our tool. We consider seven events in the log file:

1. ```
   JS.applet(‘‘Insertions’’,0).
     member(‘‘NewPanelSfondo2’’).
     multiClick(124,93,16,1);
   ```

   The tool detects that the user has selected an area that does not correspond at any interaction technique, it provides the position where it occurred, in this case it corresponds at the name of the town. Probably the user thought it was selectable and tried to receive some information about it,

2. ```
   JS.delay(7310);
   ```

   Between each couple of actions an indication providing the time passed among their occurrence is provided in millisecond. We will not report the other occurrences of similar actions.

3.
```
JS.applet(''Insertions'',0).
  button(''Sculptors'').
  typeString(''Bodin'');
```

   In this case the user started to type the name of a sculptor (*TypeValue* task in Figure 2) without selecting first the related field, our tool detects a precondition error

4.
```
JS.applet(''Insertions'',0).
  member(''java.awt.TextField'',0).
  multiClick(0,8,16,1);
```

   The user has probably understood now the type of error performed and s/he has now correctly selected a field (*SelField* task) that can receive text input

5.
```
JS.applet(''Insertions'',0).
  member(''java.awt.TextField'',0).
  typeString(''Bodini Floriano'',0,0);
```

   The user has now provided correctly the name of a sculptor ("Bodini Floriano"), performing the *TypeValue* task

6.
```
JS.applet(''Insertions'',0).
  button(''Periods'').buttonPress();
```

   Now the user has selected the button corresponding at the request to show the list of historical periods considered (*SelPeriods* task)

7.
```
JS.applet(''Insertions'',0).
  member(''NewPanelSfondo2'').
  member(''java.awt.List'',2).
  select(4,''XVIII sec.'');
```

   Finally the user has selected the historical period of interest (*SelValue* task) from the list dynamically shown by the application.

7 The Results Provided

Our method can provide a wide variety of results that can be useful for the evaluator. They can be related to both single sessions and groups of sessions. It is possible to obtain some general information on the user sessions (such as duration, number of tasks failed and completed, number of errors, number of scrollbar or windows moved, see Figure 5), more detailed information for the tasks considered, and some graphical representations of such results. When tasks are counted we consider all the tasks in the ConcurTaskTrees specification both basic tasks and higher levels tasks.

The more detailed information about the tasks include:

- The display of the accomplished tasks and how many times they are performed. The frequency of the tasks can be useful when deciding the layout of the interface. For example, it is often preferable to highlight the interaction techniques associated with frequent tasks.

- 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 display of the tasks the user never tried to perform, this information can be useful to identify parts of the user interface that are either useless or difficult to achieve for the users; this result is more difficult to obtain with other approaches based on observations.

- Display of all the errors divided into precondition errors and others.

- The display of the task patterns found (specific sequences of tasks) among the accomplished tasks (see Figure 6). The presentation shows first the frequency and next the pattern, and orders them by frequency. Patterns are useful to identify sequence of tasks frequently performed by users. This information can be useful to try to improve the design so as to speed-up the performances of such tasks.

- 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 it at a later time.

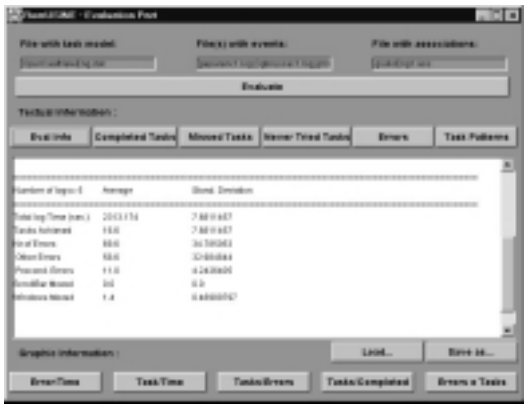

Figure 5: An example of general information.

Figure 6: An example of task patterns detected.

The different graphs, showing the data from the evaluation in different manners, are:

- The *Tasks/Time* chart graph with the tasks on the x-scale and how long they took to perform on the y-scale (see Figure 7). For each task the related bar chart highlights the fastest, the slowest and the average performance in the group of sessions considered.
- The *Errors/Time* graph with the number of errors on the y-scale and the time on the x-scale.
- The *Tasks/Errors* chart graph containing the number of precondition errors associated with each task.
- The *Tasks/Completed* chart graph containing the number of times the tasks were performed.
- The *Errors & Tasks* pie chart containing the different types of errors and their percentage, and another containing the number of tasks accomplished, missed and never attempted.

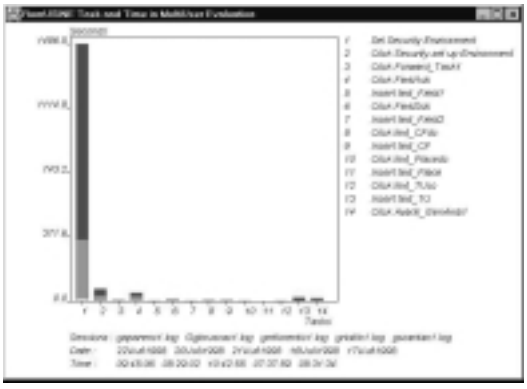

Figure 7: A diagram indicating task performances.

It is possible to provide all this information related to a single user session or to groups of user sessions. The utility to apply the tool to groups of sessions is that in this way it is possible to immediately identify if in any session there was some abnormal behaviour. For example, a task that was performed in a long time just because the user was interrupted by external factors during its accomplishment. For example, in Figure 7 evaluators can notice that the first task has a particular long performance. Then they can interactively select the task and receive information on each session concerning that task (Figure 8). In this case they can see that there is only one session during which the task required a particularly long time of performance thus indicating that there were problems only for one user and not for the others.

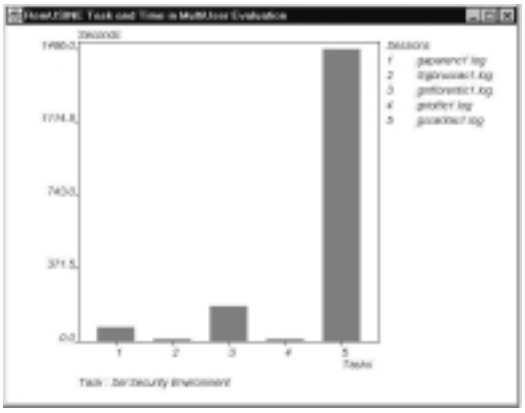

Figure 8: Representation of time performance of a selected task for each session.

8 How to Interpret the Results

The method proposed has been applied to three case studies. The first concerned the evaluation of an application for making cinema reservations. The evaluation was performed by considering eighteen computer science students.

We then applied the method in a large software company developing an application for supporting management of tax declarations and their transmission to the national centre after performing some controls. The third case study considered is a hypermedia museum application. They are all Java applications, even if the RemUSINE method can be applied to applications written in other languages.

The case studies performed showed that it is possible to find various causes for the errors detected. Possible examples are:

- *The task model supported by the user interface imposes temporal constraints too rigid for the conceptual task model of the user;* for example if a movie database application allows only to select a cinema and next the movies projected in the selected cinema, it can be too limiting for users who may want to select first the list of movies available and then ask where they are projected. This can happen, for example, when people first decide a time when to go to the cinema and thus they want to see what movies are available at that time and only after that where a specific movie is projected.

- *The user interface does not provide enough guidance on how a task should be performed;* for example the user can try to send a request to an application without having specified enough fields in the request form because the user interface does not highlight sufficiently what the mandatory fields are.

- *A label is not sufficiently explicative;* for example in some user interfaces there are buttons with a More Info label but it is not clear what the topic related to the more info is.

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 analysing user interactions.

In our experience our method has showed to be particularly suitable for applications that can be used in centres located in various remote sites because it does not require movements of users or evaluators and at the same time allows an evaluation that takes into account the real behaviour of users in their work place.

9 Evaluation of RemUSINE

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 completed).

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 task models is in the design phase (Wilson et al., 1993) 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.

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 analysing all the user tests of that application. Thus it is a limited additional effort.

On the other hand, it is possible to achieve more reliable evaluations of the application as no action to be analysed is omitted because we can automatically detect their occurrence. This is not possible with other approaches. For example, in one case study we noticed that video-based observations were not able to detect some users' clicks.

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 sequentially follow each user and analyse his/her behaviour manually. This implies great effort in terms of time from both the user and the observer and lower 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 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

that can be limited to an additional analysis of the parts of the user interface that our method finds problematic.

10 Conclusions

In this paper we have presented a new method for supporting remote usability evaluation. It is based on a model-aided analysis of the log of user events. The purpose of the analysis is to give evaluators a lot of information useful for identifying problematic parts of the user interface and possible suggestions to improve it. Consequently the effort required to the evaluators decreases so as the overall costs of the evaluation phase especially when many users sessions are considered.

References

Byrne, M., Wood, S., Noi Sukaviriya, P., Foley, J. & Kieras, D. (1994), Automating Interface Evaluation, *in* B. Adelson, S. Dumais & J. Olson (eds.), *Proceedings of CHI'94: Human Factors in Computing Systems*, ACM Press, pp.232–7.

Card, S. K., Moran, T. P. & Newell, A. (1983), *The Psychology of Human–Computer Interaction*, Lawrence Erlbaum Associates.

Hartson, H. & Gray, P. (1992), "Temporal Aspects of Tasks in the User Action Notation", *Human–Computer Interaction* **7**(***NUMBER***), 1–45.

Hartson, R., Castillo, J., Kelso, J., Kamler, J. & Neale, W. (1996), Remote Evaluation: The Network as an Extension of the Usability Laboratory, *in* G. van der Veer & B. Nardi (eds.), *Proceedings of CHI'96: Human Factors in Computing Systems*, ACM Press, pp.228–35.

Jeffries, R., Miller, J., Wharton, C. & Uyeda, K. (1991), User Interface Evaluation in the Real World: A Comparison of Four Techniques, *in* S. P. Robertson, G. M. Olson & J. S. Olson (eds.), *Proceedings of CHI'91: Human Factors in Computing Systems (Reaching through Technology)*, ACM Press, pp.119–24.

John, B. & Kieras, D. (1996), "Using GOMS for User Interface Design and Evaluation: Which Technique?", *ACM Transactions on Computer–Human Interaction* **3**(4), 287–319.

Lecerof, A. & Paternò, F. (1998), "Automatic Support for Usability Evaluation", *IEEE Transactions on Software Engineering* **24**(10), 863–88.

Nielsen, J. (1993), *Usability Engineering*, Academic Press.

Paternò, F. (1999), *Model-Based Design and Evaluation of Interactive Applications*, Springer-Verlag.

Wilson, S., Johnson, P., Kelly, C., Cunningham, J. & Markopoulos, P. (1993), Beyond Hacking: A Model-based Approach to User Interface Design, *in* J. Alty, D. Diaper & S. Guest (eds.), *People and Computers VIII (Proceedings of HCI'93)*, Cambridge University Press, pp.217–31.

Human–Computer Interaction — INTERACT'99
Angela Sasse and Chris Johnson (Editors)
Published by IOS Press, © IFIP TC.13, 1999

Author Index

Human–Computer Interaction — INTERACT'99
Angela Sasse and Chris Johnson (Editors)
Published by IOS Press, © *IFIP TC.13, 1999*

Keyword Index