User Interface Evaluation When User Errors May Have Safety-Critical Effects

Fabio Paternò & Carmen Santoro

CNUCE-C.N.R., Via G. Moruzzi 1, 56010 Ghezzano Pisa, Italy

{F.Paterno, C.Santoro}@cnuce.cnr.it

Abstract: Interactive safety-critical applications have specific requirements that cannot be completely captured by traditional evaluation techniques. In this paper we discuss how to perform a systematic inspection-based analysis to improve both usability and safety aspects of an application. The analysis considers a system prototype and the related task model and aims to evaluate what could happen when interactions and behaviours occur differently from what the system design assumes. We discuss an application of this method to a case study in the air traffic control domain.

Keywords: Model-based evaluation, safety and usability, task models, inspection-based evaluation

1 Introduction

The research area of model-based design and evaluation of interactive applications (Paternò, 1999) aims at identifying models able to support design, development, and evaluation of interactive applications. Such models highlight important aspects that should be taken into account by designers. Various types of models, such as user, context, and task models have proved to be useful in the design and development of interactive applications. Task models describe activities that have to be performed so that user's goals are attained. A goal is a desired modification of the state of an application. The use of task analysis and modelling has long been applied in the design of interactive applications. However, less attention has been paid to their use to support systematic usability evaluation. To this end, it is important to have task models described by flexible and expressive notations with precise semantics able to represent the different ways to perform tasks and the many possible temporal and semantic relationships among them. This allows designers to develop systematic methods able to indicate how to use the information contained in the task model for supporting the design and evaluation of the user interface.

There are various approaches that aim to specify tasks. They differ in aspects such as the type of formalism they use, the type of knowledge they capture, and how they support the design and development of interactive systems. In this paper we consider task models that have been represented using the ConcurTaskTrees notation (Paternò, 1999).

In ConcurTaskTrees activities are described at different abstraction levels in a hierarchical manner, represented graphically in a tree-like format (see Figure 1 for an example). In contrast to previous approaches, such as Hierarchical Task Analysis, ConcurTaskTrees provides a rich set of operators, with precise meaning, able to describe many possible temporal relationships (concurrency, interruption, disabling, iteration, option and so on). This allows designers to obtain concise representations describing many possible evolutions over a user session. The notation also gives the possibility of using icons or geometrical shapes to indicate how the performance of the tasks is allocated. For each task it is possible to provide additional information including the objects manipulated (for both the user interface and the application) and attributes such as frequency.

Automatic tools are needed to make the development and analysis of such task models easier and more efficient. A set of tools to specify task models in ConcurTaskTrees and analyse their content is publicly available at http://giove.cnuce.cnr.it/ctte.html.

Task models can also be useful in supporting design and evaluation of interactive safety-critical applications. The main feature of these systems is that they control a real-world entity and have to fulfil a number of requirements while avoiding that the controlled entity reaches hazardous states (states where there is actual or potential danger to people or the environment). There are many examples of safety-critical systems in real life (air traffic control, railway systems, industry control systems, ...). In this field specific issues arise. For instance, in safety-critical domains, sometimes user actions cannot be undone (for example, if an irreversible physical process has been activated), so the issue of *user errors* and how to design the user interface so as to avoid them, acquires a special importance. In fact, many studies have shown that accidents often are caused by a human error whose likelihood may be increased by poor design.

The goal of this paper is to discuss how task models can be used in an inspection-based usability evaluation for interactive safety-critical applications. To this end, we first introduce our approach and a method to describe how to use information contained in task models to support an exhaustive inspection-based evaluation. We then discuss an exercise in applying this method to a real case study, within a multidisciplinary team, in the air traffic control domain. Finally, we discuss this experience in terms of results and lessons learnt, providing some concluding remarks.

2 Task Models and Usability Evaluation

Task models can be useful in various phases of the design cycle. For example, it is possible to identify criteria for supporting model-based design of interactive application. They can also be useful in the evaluation phase. The work done so far in model-based evaluation has mainly aimed to support performance evaluation (such as GOMS approaches) or to use the model to support automatic analysis of user interactions (Ivory et al, 1999).

This type of evaluation is useful for evaluating final versions of applications. However, we think that task models may also help support evaluation in the early phases of the design cycle. Indeed, inspection-based methods are often applied to evaluate prototypes. They are less expensive than empirical testing and one advantage is that their application at least decreases the number of usability problems that can be detected by the final empirical testing.

A number of inspection-based evaluation methods have been proposed. In heuristic evaluation (Nielsen, 1993) a set of general evaluation criteria (such as visibility of the state of the system, consistency, avoid providing useless information, ...) are considered and evaluators have to check whether they have been correctly applied. This method heavily depends on the ability of the evaluator and many software engineers may have problems to understand how to apply such general criteria to their specific cases. In cognitive walkthrough (Wharton et al, 1994) the evaluators have to identify a sequence of tasks to perform and for each of them four questions are asked. While this method is clear and can be applied with limited effort, in our opinion it has a limitation: it tends to concentrate the attention of the evaluator on whether or not the user will be able to perform the right interactions. Little attention is paid on what happens if users perform errors (interactions that are not useful for performing the current task) and on the consequences of such errors. It is easy to understand how crucial this aspect is in interactive safety-critical applications where the consequences of human errors may even threaten human life.

Few works have addressed this type of problem. Reason (Reason, 1990) introduced a first systematic analysis concerning human error, including the simple slips/mistakes distinction. Human Reliability Analysis (Hollnagel, 1993) stems from the need to quantify the effects of human error on the risks associated with a system and typically involves estimating the probability of the occurrence of errors, which quickly runs into the problem of acquiring the data necessary for reliable quantification. Practical experience shows that different methods in this area often yield different numerical results, and even the same method may give different results when used by different analysts. Leveson (Leveson, 1995) introduced a set of guidelines for the design of safety-critical applications, but little attention was paid to the user interface component. Some guidelines for safe user interactions design are proposed, but they are too general to use systematically when designing the user interface. Then, research moved on to finding systematic methods for supporting design and evaluation in this area. THEA (Fields et al, 1997) uses a scenario-based approach to analyse possible issues. Formal methods (Palanque et al., 1997) have also been considered for this purpose. In (Galliers et

al., 1999) the authors propose an analysis that supports re-designing a user interface to avoid the occurrence of errors or to at least reduce their effects. The analysis is supported by a probabilistic model that uses Bayesian Belief Nets. Johnson has developed a number of techniques -see for example (Johnson, 1999)- for analysing accident reports, which can be useful to better understand the human errors that have caused real accidents.

The contribution of our method resides in the help that it provides to designers in order to systematically analyse what happens if there are deviations in task performance with respect to what was originally planned. It indicates a set of predefined classes of deviations that are identified by *guidewords* (MOD, 1996). A guideword is a word or phrase that expresses and defines a specific type of *deviation*. These types of deviations have been found useful for stimulating discussion as part of an inspection process about possible *causes* and *consequences* of deviations during user interactions. Mechanisms that aid the *detection* or *indication* of any hazards are also examined and the results are recorded.

3 The Method

In the analysis, we consider the system tasks model: how the design of the system to evaluate assumes that tasks should be performed. The goal is to identify the possible deviations from this plan. Interpreting the guidewords in relation to a task allows the analyst to systematically generate ways the task could potentially deviate from the expected behaviour during its performance. This serves as a starting point for further discussion and investigation. Such analysis should generate suggestions about how to guard against deviations as well as recommendations about user interface designs that might either reduce the likelihood of the deviation or support its detection and recovery from hazardous states.

The method is composed of three steps:

- *Development of the task model of the application considered*; this means that the design of the system is analysed in order to identify how it requires that tasks are performed. The purpose is to provide a description logically structured in a hierarchical manner of tasks that have to be performed, including their temporal relationships, the objects manipulated and the tasks' attributes.
- *Analysis of deviations related to the basic tasks*; the basic tasks are the leaves in the hierarchical task model, tasks that the designer deems should be considered as units.
- *Analysis of deviations in high-level tasks*, these tasks allow designer to identify *group of tasks* and consequently to analyse deviations that involve more than one basic task. Such deviations concern whether the appropriate tasks are performed and if such tasks are accomplished following a correct ordering.

It is important that the analysis of deviations be carried out by interdisciplinary groups where such deviations are considered from different viewpoints in order to carry out a complete analysis. The analysis follows a bottom-up approach (first basic tasks, and then high-level tasks are considered) that allows designers first to focus on concrete aspects and then to widen the analysis to consider more logical steps.

We have found useful to investigate the deviations associated with the following guidewords:

- *None*, the unit of analysis, either a task or a group of tasks, has not been performed or it has been performed but without producing any result. None is decomposed into three types of deviations depending on why the task performance has not produced any result. It can be due to a *lack of initial information* necessary to perform a task or because *the task has not been performed* or because it has been performed but, for some reason, *its results are lost*. An example is when the user updates a system and for some reason the results of this updating are not presented on the screen.
- *Other than*, the tasks considered have been performed differently from the designer's intentions specified in the task model; in this case we can distinguish three sub-cases: less, more or different. Each of these sub-cases may be applied to the analysis of the input, performance or result of a task, thus identifying nine types of deviations (*less input, less performance, less output, other than input, other than performance, other than output, more input, more performance, more output*). An example of the less input sub-case is when the user sends a request without providing all the information necessary to perform it correctly. Thus, in this case the task produces some results but they are likely wrong results.
- *Ill-timed*, the tasks considered have been performed at the wrong time. Here we can

distinguish at least when the performance occurs *early or late* with respect to the planned activity.

In addition, for each task analysed it is possible to store in one table the result of the analysis in terms of the following information:

- *Task*, indicating the task currently analysed;.
- *Guideword*, indicating the type of deviation considered.
- *Explanation*, explaining how the deviation has been interpreted for that task or group of tasks;
- *Causes*, indicating the potential causes for the deviation considered and which cognitive fault might have generated the deviation;
- *Consequences*, indicating the possible effects of the occurrence of the deviation in the system;
- *Protection*, describing the protections that have been implemented in the considered design in order to guard against either the occurrence or the effects of the deviation;
- *Recommendation*, providing suggestions for an improved design able to better cope with the considered deviation;

We think useful to classify the explanation in terms of which phase of the interaction cycle (according to Norman's model) can generate the problem:

- *Intention*, the user intended to perform the wrong task. An example of intention problem is while controllers aim to solve an air traffic conflict to obtain a safer state they decide to increase the level of an aircraft but in this manner they create a new conflict.
- *Action*, the task the user intended to perform was correct but the actions identified to support it were wrong.
- *Execution*, the performance of an action was wrong, for example the controller selects a wrong aircraft because of a slip.
- *Perception*, the user has difficulties in perceiving or correctly perceiving the information that is provided by the application. A perception problem is when the controller finds difficult to read an enriched flight label of an aircraft placed within an aerodrome area crowded of planes (overlapping labels).
- *Interpretation*, the user misinterpreted the information provided by the application. An interpretation problem is when a controller believes that an aircraft is under his responsibility whereas actually the other controller is in charge of it.
- *Evaluation*, in this case the controller has perceived and interpreted correctly the information but it is wrongly evaluated, for example the controller detects an air traffic conflict that does not exist.

Tables can be used as documentation of a system and its design rationale, giving exhaustive explanation of how an unexpected use of the system has been considered and which type of support has been provided in the system to handle such abnormal situations. Many types of strategies can be followed for this purpose: these range from techniques aiming to prevent abnormal situations, to others that allow such situations, but inform the user of their occurrence in order to either mitigate or aid in recovering from potential problems (if any). To better explain how the method works, we consider an example of both basic task and high-level task.

Table 1 presents an example of the analysis of a basic task in an air traffic control case study. The task considered is *Check deviation* (the user checks whether aircraft are following the assigned path in the airport). The class of deviation considered is *None*. First, we have to identify the information required to perform the task. In this case it is the state of the traffic in the airport and the path associated with the aircraft. If we consider the *No input* case, we can note that it has different causes, both generating perception problems: either a system fault has occurred or the controller is distracted by other activities. In any case, the consequence is that the controller has not an updated view of the air traffic.

The protection in the current system changes accordingly, if the system is broken then the controller has to look through the window in order to check the state of the traffic. If the controller is distracted then pilots, especially if they perceive that something abnormal is occurring, may contact the controller. The recommendations for an improved system change depending on the case considered: duplication policy should be followed against system faults whereas warning messages should be provided in case of aircraft deviating from the assigned path. Slightly different possibilities are considered in the other sub-cases. For instance, in the *No performance* case we consider when the information is available but for some reason the task is not performed: the controller is distracted or over confident, so he does not interpret correctly the information. In the *No output* case we have that the task is performed but its results are lost, e.g. the controllers find a deviation but they forget it because they are immediately interrupted by another activity.

Task: Check Deviation				Guideword: None
Explanation	Causes	Consequences	Protections	Recommendations
No input: The controller does not have information concerning the current traffic **No performance**: The controller has the information but he does not check it carefully **No output**: The controller finds a deviation but immediately forgets about it	The system is broken *Perception* problem Controller is distracted or interrupted by other activities Controller is distracted or overconfident *Interpretation* problem Controller is interrupted by other activities *Intention* problem	The controller has not an updated view of the air traffic	The controller looks at the window to check the air traffic Pilot calls controller to check the path Red line in case of runway incursion	Duplication of communication systems Provide an automatic warning message only when an aircraft is deviating from the assigned path.

Table 1: Example of analysis of task deviations.

When considering not basic tasks, the deviations should be applied in a slightly different manner although we can still use similar tables to store the results of the analysis. In fact, in these cases, the interpretation and the application of each guideword to a higher-level task has to be properly customised depending on the temporal relationships existing between its subtasks. For example, consider a higher-level task which has all its basic tasks connected by the sequential enabling operator ([]>> or >> depending on whether information is exchanged between the tasks or not), for example *Build path with automatic suggestions* task in Figure 1. Because of the temporal relationship that relates the first left subtask to the others, a lack of input for the first basic task means a lack of input for the whole parent task. Therefore, this case (*None/No input*) of the analysis of the parent task can be brought back to the correspondent case of its first left child. The *None/no performance* case means that no lower-level subtask (*Select Automatic Modality, Show suggested solutions* and *Select solution*) has been performed. The *None/No output* case is when all the subtasks have been carried out, but no output has been produced at the end. Referring to the aforementioned *Build path with automatic suggestions* task in Figure 1, this case can be brought back to the *None/no output* case of the *Select solution* task.

For the other guidewords, the reasoning change accordingly. For example, the *Less* guideword applied to the *Build path with automatic suggestions* task means that one associated sub-task has not been carried out. For example, the user forgets to perform the last task (*Select solution*) after executing the first

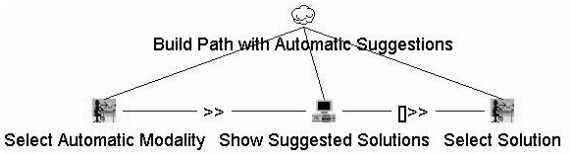

Figure 1: An example of high-level task.

two subtasks (*Select Automatic Modality* and *Show suggested solutions)*. The *More* case might arise because of an additional, not expected performance of one subtask (e.g.: one task is executed more than one time). The *Different* case is when a different temporal relationship is introduced within the subtask (e.g.: the subtasks are concurrently performed instead of being executed sequentially) or wrong subtasks are performed. The analysis of *Early* and *Late* guidewords should consider the parent task as a whole, inquiring about the impact on the system of all the situations when a too early/late performance of the parent task occurs.

Of course, with different arrangement of temporal operators the reasoning has to be appropriately customised. It is beyond the scope of this paper to give the reader an exhaustive analysis of all the possible cases treated. Instead, what has to be

emphasised is that, depending on the specific decomposition of a higher-level task into lower-level tasks, the application of the guidewords has to be adapted on the basis of the particular temporal relationships specified.

4 Organisation of the Evaluation

The evaluation exercises should be carefully organised. It is important to involve, besides the evaluators, at least some software developers and real end-users as well. The participants should be introduced to the method so that they can understand the structure of the exercise and the reasons for it, and they should spend some time to understand the features of the prototype.

During the exercise the prototype (at whatever level of refinement it is) should be available and the evaluators should have the task model available in order to have a representation helping them to systematically identify what should be analysed. In order to simplify the exercise it is not necessary that end users follow the details of the notation used for specifying the task model. It is sufficient that the model is used by the evaluators who analyse it to decide the questions or issues to raise. It can be useful to have an audio record of the session that can be considered later on to check some parts of the discussion.

During the session, evaluators should drive the discussion raising questions following the task model and the list of deviations. Often in the discussion it is useful to explain which presentation elements of the user interface support the logical task considered. In addition, for each task it is important to clearly identify what information is required to perform it and what the result of its performance is.

It is not necessary to include all the possible cases in the tables, such as Table 1, usually the description of the more important ones is sufficient to have an exhaustive analysis. Tables can take some effort to fill them and such effort is justified especially when it is required to have a systematic documentation of the system and design rationale.

5 Our Experience

We have applied the method to a prototype for the air traffic control in an aerodrome. The main purpose of the system is to support data-link communications to handle aircraft movement between the runways and the gates.

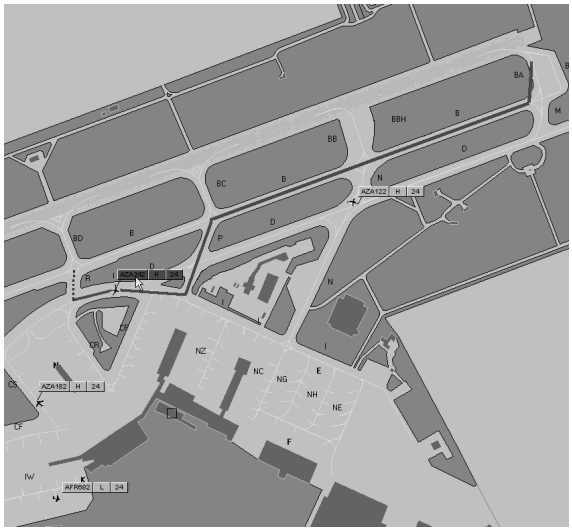

Figure 2: The user interface of the prototype evaluated.

Data link is a technology allowing asynchronous exchanges of digital data coded according to a predefined syntax. It complements the current communication technology that is based mainly on radio communications and observations from the window of the control tower. The support of this new technology is particularly useful in case of bad atmospheric conditions. Thus, controllers can interact with graphical user interfaces showing the traffic within an airport in real-time (see for example Figure 2) and messages received from pilots. In addition, in this type of application controllers can compose messages by direct manipulation techniques.

In the airport there are two main controllers: the “ground”, who handles the aircraft on the taxiways until they reach either the beginning of the runway (for departing aircraft) or the assigned gate (for arriving aircraft), and the “tower”, who decides on take-off and landing of aircraft. In the considered system, the controllers have enriched flight labels available on their user interface instead of traditional paper strips commonly used in current air traffic control centres. The enriched flight labels show permanently just essential information concerning flights (standard mode) and can interactively enlarge to show additional information (selected mode) and shrink again. They are different depending on the specific controller. For instance, the enriched labels shown in standard mode on the ground’s user interface include information about the flight identifier, the category, and the assigned runway.

When a label is selected, an extended set of information is given including the path assigned to a

flight (which can be both textually and graphically represented), the current velocity of the aircraft, and other useful data for the ground controller (Figure 3).

We carried out the exercise with a multidisciplinary team in which software developers, final users (air traffic controllers in this case) and experts in user interface design were involved. Having a multidisciplinary team is useful because a problem is seen from different viewpoints and perspectives and everyone gives its own contribution based on his own knowledge, skills and expertise. In this way, some problems that could be overestimated are brought back into the correct terms and seen in the right perspective and, on the other hand, there is no risk of underestimating other aspects.

During these exercises, many interesting issues arose. We give examples of some of them indicating the tasks and the deviations that raised the issues in order to allow understanding how the method was useful to identify them. We also show how an analysis of the source of the problem can help identify new solutions for an improved design.

An example is related to the controller's task of checking if the current state of an aircraft conforms with the expected one, especially in terms of current/planned positions in the aerodrome area. If this task is not performed by the controller (because of a high level of workload, or a controller's memory fault, or a system fault) the existing protection limits its occurrence only to truly hazardous situations as a runway incursion is. In this case a red line connecting the involved aircraft is highlighted on the controller's user interface. However, this protection is not sufficient yet: whenever it is really crucial to attract the attention of the controller a visual warning is not always the most appropriate technique, especially in an environment where the user has to control a lot of visual tools and media. An audio warning can be more suitable, especially when it is necessary to convey information about the *urgency* of the situation. Earcons can be a useful improvement to the current prototype because support flexible calibration of their aural attributes. However, when the controllers are already aware of the hazard, they feel very annoyed by useless acoustic warnings. Thus, it is important to allow them to disable such alarms (usually in situations of high traffic). Controllers found such alarms particularly useful in situations of low traffic when they tend to get distracted and overconfident.

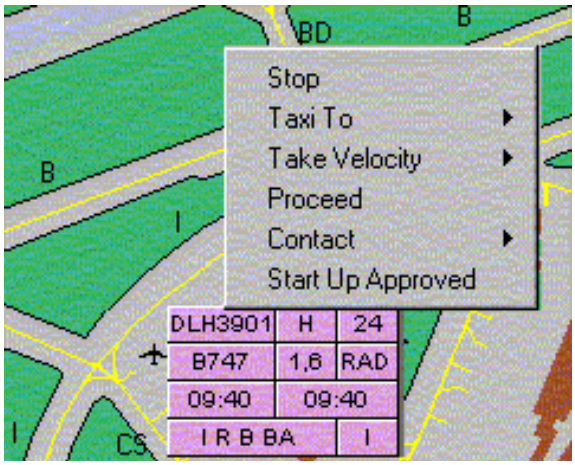

Figure 3: The menu supporting the *Stop* command.

In the prototype it was already possible for the controller to zoom in or out the aerodrome map in order to have a more detailed view of some parts of the aerodrome. However, it is possible that some conflicts occur in the part of the airport currently out of the controller's view (*No input* case of the *Check Deviation* task) because the controller has just zoomed in a different part of the aerodrome map. In this situation, the possibility of having a button to return to the default view is not sufficient to make the controller aware of some hazardous situations in time. Thus, the suggestion was to have permanently displayed the global view of the aerodrome map (for evident safety goals) and, on request, to activate a separate window highlighting a specific part of the aerodrome.

Another deviation considered was associated with the *Send clearance* task, *No performance* deviation. The causes can be either a system's fault or a pilot distracted. The protection against similar deviations in the current system can be afforded by detecting the lack of reaction from the pilot after a clearance has been sent. However, we suggested that automatic acknowledgement of the fact that the clearance has reached the pilot's system is very important in such highly safety-critical system.

Another problem occurs when controllers can choose among a set of clearances to send. The menu supporting this choice among multiple tasks (see Figure 3) considers them all in the same manner. However, in case of error their consequences can considerably differ. In particular, in some cases the stop task is particularly important as controllers may detect that one aircraft is going to crash somewhere. In this case a slip error in the selection of the stop element would be really dangerous. In the current menu the stop element is located in the best position: it is the first item of the menu, thus the fastest to be

found and with fewer possibilities to be erroneously confused with another element. Nevertheless, better highlighting this command by appropriated graphical attributes (such as size and colour) would be even more effective.

Another problem was still associated with the task of stopping an aircraft during take-off. In the prototype, it is always possible to stop an aircraft taking-off because the related command (Stop) is always available on the user interface. However, in real situations, the command "Stop" should be no longer enabled once the aircraft has started the take-off because trying to abort such action at that moment is too late ("ill-timed" performance). Thus a different temporal relationship is required when composing this task with the tasks related to sending other clearances. This is a small example to show how this analysis can be useful also to discover limitations in the system task model itself.

6 Conclusions

In this paper we have discussed how task models and guidewords can support a systematic inspection-based usability evaluation for interactive safety-critical applications. The combination of basic and high-level tasks with the many types of deviations considered allows designer to address a broad set of issues. This can be useful to analyse how the system design supports unexpected deviations in task performance. This type of analysis is particularly important in interactive safety-critical systems where a user error can even threaten human life. We have also discussed the application of this method to a prototype for air traffic control.

Our experience has shown the effectiveness of the method despite some social constraints that often occur in software development enterprises (software developers tend to defend every decision taken, users tend to digress in the teamwork, time pressure). The systematic analysis with multidisciplinary teams (including real users) provides useful and thorough support to detect potential safety-critical interactions and improve the design.

7 Acknowledgements

We gratefully acknowledge support from the European Commission and our colleagues in the (http://giove.cnuce.cnr.it/mefisto.html) MEFISTO project for useful discussions.

References

Fields, R.E., Harrison, M.D. and Wright, P.C. (1997). THEA: Human Error Analysis for Requirements Definition. University of York, Dept. of Computer Science, Technical Report YCS-97-294.

Galliers, J., Sutcliffe, A., Minocha, S.(1999), "An Impact Analysis Method for Safety-Critical User Interface Design", ACM Transactions on Computer-Human Interaction, Vol.6, N.4.

Hollnagel, R.(1993), *Human Reliability Analysis —Context and Control*. Academic Press.

Ivory, M., Hearst, M. (1999), Comparing performance and usability evaluation: New methods for automated usability assessment. 1999. Report available at http://www.cs.berkeley.edu/~ivory/ research/web/papers/pe-ue.pdf

Johnson, C., and Botting, R. (1999), Reason's Model of Organisational Accidents in Formalising Accident Reports, *Cognition, Technology and Work*, 1(3), 107-118.

Leveson, N.G. (1995), *Safeware: System Safety and Computers —A Guide to preventing accidents and losses caused by technology*, Addison Wesley.

MOD (1996) HAZOP Studies on Systems Contaning Programmable Electronics. UK Ministry of Defence Interim Def Stan 00-58, 1996, Issue 1. Available at http://www.dstan.mod.uk/dstan_data/ix-00.htm

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

Palanque P., Bastide R, Paternò F. (1997), Formal Specification as a Tool for Objective Assessment of Safety-Critical Interactive Systems, *Proceedings INTERACT' 97*, Chapman&Hall, pp.323-330.

Paternò, F. (1999), *Model-based Design and Evaluation of Interactive Applications*, Springer Verlag, ISBN 1-85233-155-0.

Reason, J. (1990), *Human Error*. Cambridge University Press.

Wharton, C., Rieman, J., Lewis, C., and Polson, P.(1994), The Cognitive Walkthrough: A Practitioner's Guide, *in* J. Nielsen and R.L. Mack (eds.), *Usability Inspection Methods*, John Wiley & Sons.