Interacting with Computers 12 (2000) 565–586

Interacting with Computers

www.elsevier.com/locate/intcom

An analysis of errors in interactive proof attempts

S. Aitken[a,*], T. Melham[b]

[a]Artificial Intelligence Applications Institute, University of Edinburgh, Edinburgh, Scotland EH1 1HN, UK
[b]Department of Computing Science, University of Glasgow, Glasgow, Scotland G12 8QQ, UK

Received 12 May 1999; received in revised form 19 October 1999; accepted 22 October 1999

Abstract

The practical utility of interactive, user-guided, theorem proving depends on the design of good interaction environments, the study of which should be grounded in methods of research into human–computer interaction (HCI). This paper discusses the relevance of classifications of programming errors developed by the HCI community to the problem of interactive theorem proving. A new taxonomy of errors is proposed for interaction with theorem provers and its adequacy as a usability metric is assessed experimentally. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Interactive proof attempts; Interaction environments; Human–computer interaction

The past decade has seen many advances in using computers to do logical proofs and in the deployment of this technology in formal methods applications. Fully automatic methods, such as equivalence checking and model checking for hardware, are already being used in industry [1]. Although these automatic methods are important, they are also limited in the scope of problems they can address. A complement to automatic tools are computer systems for *interactive* proof-construction, where the user helps guide the system towards proof discovery and construction.

Numerous interactive proof-construction tools have been developed for various logics and styles of reasoning, and some systems have reached a high state of maturity with a significant user base in academia and industry. Prominent examples include Isabelle [2], HOL [3] and PVS [4]. Many application experiments have been done with these tools, and the results of this research have begun to reach industry through, for example, the notable recent achievements in hardware verification using PVS [5] and the use of HOL at several industrial sites [6,7].

But despite these encouraging case-studies, interactive theorem-prover technology has

* Corresponding author. Tel.: +44-131-650-2738; fax: +44-131-650-6513.
E-mail addresses: stuart@aiai.ed.ac.uk (S. Aitken); tfm@dcs.gla.ac.uk (T. Melham)

still not attained really widespread use in actual industrial practice. One commonly-cited obstacle has been the poor quality of interaction environment historically provided by most systems. In response to this, several projects have been undertaken to develop good interfaces for specific systems. Notable examples include the XIsabelle GUI for the Isabelle theorem prover [8], the CHOL [9] and TkHOL [10] graphical interfaces for the HOL system, the CtCoq interface for the Coq theorem prover [11,12], and the graphically driven Jape system [13]. Another line of work is represented by the emacs interface to PVS and the more generic Proof General interface [14], which is also based on emacs and has been customised for several theorem-proving systems.

The results of these efforts have been encouraging, although many interaction problems remain unsolved. But regardless of the success or failure of these experiments, designers of interfaces to proof systems have rarely drawn explicitly upon research in HCI or employed user-centred design methods. This paper describes work undertaken as part of an initiative to rectify this lack of analysis.

The ITP Project at the University of Glasgow aimed to apply the principles and methods of HCI to the analysis and design of interaction with theorem-proving systems. We have applied Norman's activity model [15], task modelling, and knowledge modelling techniques to gain a wide understanding of user–system interaction. We believe that new proof tools should solve real usability problems, hence our research is initially concerned with identifying and understanding usability problems. Others are investigating the formal specification of the interface and cognitive mechanisms [16,17].

In this article, we focus on user errors in interactive proof construction. Our starting point is the observation that interactive theorem provers are often embedded in programming environments, in such a way that proof generation is achieved through function application and computation over data objects representing theorems (although the user need not think of the activity in these terms). Using such a system has much in common with programming; syntactically complex objects are constructed and manipulated, and the construction is a piecemeal process guided by plans at a semantic level.

Programming and properties of programming languages have been extensively studied in HCI, and environments for interactive languages, including Prolog and Lisp, have been proposed and evaluated [18–21]. But the usability of interactive theorem provers is relatively unexplored. While studies of errors in programming appear particularly relevant to interactive theorem proving, this paper employs a more general classification of errors in computer use. We describe a specialisation of a taxonomy proposed by Zapf et al. [22] which tailors it to the theorem-proving domain. We then describe an empirical study of two groups of users, and show how the error taxonomy can be used to analyse the data. This study permits us to both validate the coverage of the proposed error taxonomy, and to draw some conclusions on the usability of the theorem-proving systems used. Our conclusions on the potential of the error taxonomy as a usability metric are based on these empirical findings.

As already mentioned, the potential for improving user–prover interaction through graphical displays has inspired much recent work. In this paper, however, we do not directly address issues such as the design of graphical displays for proof trees-for example as proposed by Lowe and Duncan [23] or Schubert and Biggs [24]—or the more general problem of displaying state of a program, as described in [19]. Command line interfaces to the systems studied are still quite widespread, particularly among expert users. And our work is ultimately relevant to the evaluation of both graphical and textual interfaces; for example, results of our studies provide baseline measurements which should be improved upon by better interactive environments.

1. Interactive theorem provers

In many approaches to mechanised theorem proving, a computer program is used to try to determine the truth of a proposition completely automatically, perhaps based on a decision procedure or heuristics. But this is often hopelessly impractical, and theorem provers such as HOL, PVS, Isabelle, and LP [25] have evolved that share a much more interactive approach to proof construction. Here, the user participates intimately in proof discovery and construction, supplying the theorem-proving system with key steps in the proof and relying on it to look after the small details. Proof construction is very much a piecemeal and incremental process, with the user trying out proof steps, interpreting feedback from the system, and frequently backtracking to change approach.

In this setting, the most natural approach to discovering a proof is by working backwards from the statement to be proved (called a *goal*) to previously proved theorems that imply it. This is the *backward proof* style, in which the search for a proof proceeds by exploring possible strategies for achieving a goal. For example, one might try proving a conjunctive formula $P \wedge Q$ by breaking it down into the two separate subgoals of proving P and proving Q. Repeating this process constructs a *proof tree* that represents successive stages in the decomposition into subgoals of a conjecture to be proved. The root is the original conjecture, and the leaves are the subgoals remaining to be proved. A proof attempt is successful when all the leaves are immediately provable by the theorem prover.

Many interactive theorem provers supply functions or operations, often called *tactics*, that the user can invoke to build a backward proof tree. The system maintains a *proof state*—a representation of the decomposition of the original conjecture into subgoals (possibly including behind-the-scenes information that justifies the decompositions), together with a record of which subgoals remain to be proved. Applying a tactic modifies the current proof state by reducing one of the outstanding subgoals to one or more (easier to prove) subgoals, or by simply proving a subgoal outright. The proof is finished when no unproved subgoals remain.

At the lowest level of granularity a tactic may implement a single inference rule such as Modus Ponens. In the HOL system, for example, the tactic `CONJ_TAC` decomposes conjunctive goals into their separate conjuncts. So if the goal currently of interest is displayed as `"P/\Q"` and the next command line input is

```
> e CONJ_TAC;;
```

then the goal is reduced to two subgoals, displayed as `"P"` and `"Q"`. The expressions in quotation marks are object logic formulae, and the tactic name '`CONJ_TAC`' is part of the command language. Most systems (including HOL) also make available much more powerful tactics that can do arbitrary amounts of deduction.

In tactic-based theorem provers, one can generally distinguish two different languages, the formalism in which one proves theorems (called the *object language*) and the system's command language (called the *metalanguage*, because it is used to manipulate the object language). The metalanguage includes expressions denoting tactics, as well as constructs for combining elementary tactics together into more complex ones. Also included are commands for setting up an initial goal and inspecting the proof state.

Many systems employ the so-called *LCF approach* to theorem proving, in which the command language is a strongly-typed functional programming language and an abstract data type of theorems is used to distinguish proved facts from arbitrary propositions [26]. In HOL and Isabelle, for example, the metalanguage is the ML programming language. The most primitive interface to goal-directed proof in HOL is just an ML interpreter in which a suitable proof state has been implemented by an ML data structure and some associated functions for modifying and inspecting it have been made available. The LCF approach is not universal (e.g. PVS does not use it), but all computer-based theorem provers have some kind of metalanguage for manipulating expressions in the object logic. This two language set-up is reflected directly in the error categories proposed in the present work.

The product of a session with a theorem prover is, naturally, a set of proved theorems. Some of these may be used as lemmas in future proofs. Moreover, a proof may require definitions to be made. For example, an object-language function may be defined recursively over lists or specified by axioms. Definitions, like theorems, are data objects which can be referred to by name in the metalanguage script of proof. Making definitions and using them is a general feature of the proof activity and can be considered to be as fundamental as backward proof itself.

1.1. A model of interaction

The error analysis in this paper is based on the idea of interaction levels. We introduce this idea and the proof-as-programming paradigm before discussing errors in proof in detail in the following section. We propose that for a design-oriented description of user interfaces to theorem provers three levels can usefully be distinguished:

A concrete interaction level: At this level are actions on input devices and the perceptual characteristics of display objects. Typing characters or clicking on a button are examples of concrete interactions.

An abstract interaction level: At this level are the shared objects and operations in terms of which information is communicated from user to system. Typically, these are visual(isable) objects and the operations upon them, but abstracted away from details of their physical form. Examples include diagrams, structured text, visualised lists.

A logical level: This is a description solely in terms of logical concepts. In proof, the logical is the level of mathematical logic as this is the application domain. A description of a proof which is made using terms such as *case split* or *induction* is made at the logical level.

Each level of abstraction is self-contained, in the sense that a full description of the activity can be framed within each level. One can instruct a user to carry out a proof entirely in terms of articulations of devices or, assuming that the user knows, or can guess or learn, the representation, in terms of the higher level abstractions. However, a full description, even at a given level, must include operations which extend beyond those in which there is a flow of information between the parties to the interaction. That is, there are operations which belong solely to one party or the other and which are needed to form a complete picture of the activity.

A multi-level view of interaction has been used as the basis for previous explanations of human-computer interaction, albeit never applied specifically to the domain of interactive theorem proving. Nielson [27] presents a linguistic account of interaction by asserting that the levels correspond to the lexical, syntactic and semantic levels of linguistic activity. Norman [15] asserts that translations from one level to another form the basis for important potential interaction problems (the so-called gulfs of execution and evaluation) which arise when a user is not able to perform the transformation from one level to another. This characterisation of interaction forms the basis for an explanation of the potential advantages of direct manipulation [28], viz. it reduces the complexity of the inter-level mappings and metaphorical representations.

1.2. Proof-as-programming

In Aitken et al. [29] we identify three views of proof activity, each representing an alternative instantiation of the three level interaction model. We view interaction with HOL as *proof-as-programming*, which recasts proof problems, as defined in the logical domain, as programming problems. Fig. 1 illustrates the levels of abstraction as they are instantiated in the proof-as-programming view.

The view that proof is programming regards tactic proofs as programs to compute theorems. We develop tactic proofs piecemeal, by applying individual tactics that break the goal down into simpler and simpler subgoals. But once we have found the proof we compose the tactics into a monolithic and complete proof strategy—a functional program which we can execute to prove the desired theorem from scratch. The final product of our activity is a program; and so goal-directed theorem proving using tactics is a specialised kind of programming activity.

It does not follow from the above that users think of proof construction as programming. Rather, one would expect that, given a successful user interface, users would be relatively unaware of writing and executing parts of a program. This is an important point. Programming is not a metaphor for proof construction; instead, programming is the medium through which proofs are constructed and expressed. One would expect that a descent from thinking at the logical level to planning and evaluating at the abstract interaction level would occur if there is a breakdown in the interaction (i.e. some required information is not available or an operation not known). An advantage of proof as programming is the fact that when breakdown occurs because of the lack of a representation in the abstract interaction domain (e.g. there is no tactic which represents some desired goal decomposition), one can bridge the representational gap without switching domains.

Construction and execution of program *texts* is the fundamental organising concept of the proof-as-programming view. Tactics and the functions that combine them are the medium through which the user interactively explores possibilities and constructs a

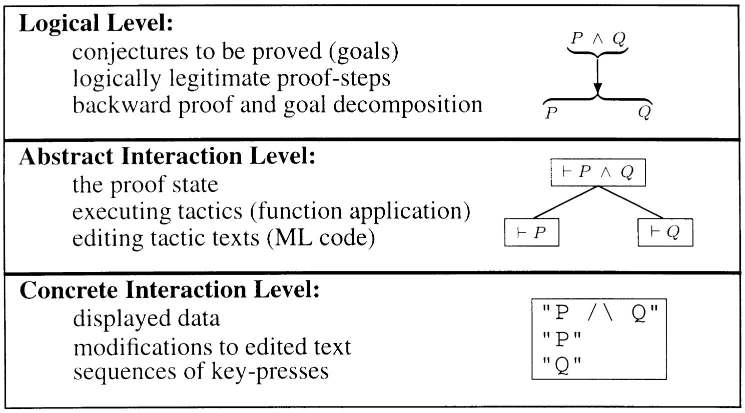
Logical Level:
conjectures to be proved (goals)
logically legitimate proof-steps
backward proof and goal decomposition
P ∧ Q
P
Q
Abstract Interaction Level:
the proof state
executing tactics (function application)
editing tactic texts (ML code)
⊢ P ∧ Q
⊢ P
⊢ Q
Concrete Interaction Level:
displayed data
modifications to edited text
sequences of key-presses
"P /\ Q"
"P"
"Q"

Fig. 1. Proof-as-programming.

proof; they also constitute program texts, which are executed to extend the proof state and are kept as a permanent record of the proof.

Because there is a clear relationship between proof activity and programming, we believe that an assessment of errors in interactions with HOL can be based on similar classifications to those describing programming errors. The refinements we propose are specific to interactive theorem provers, but do not describe errors which are peculiar to the HOL prover alone. Interaction with Isabelle is also based on tactic application at the ML command line. The amount of purely programming activity performed by users is somewhat less than in HOL, but we can still describe proof in Isabelle by the term proof-as-programming.

2. Errors in interactive proof

Errors in programming and computer use have been studied for many purposes, including comparing programming language design, validating models of user performance and cognition, and evaluating interactive systems. Our intention is to identify usability problems associated with the command line interaction style typical of interactive provers. This style of interaction is common to many interactive systems which operate in domains where direct manipulation techniques are inappropriate.

The analysis presented here is relevant to systems where the user can interactively modify the proof state by entering commands. Further, we account for the possibility that user can utilise named objects in the system and define new objects. We specifically distinguish commands that perform logical inference from those which do not, and treat the object and metalanguages separately. These elements of our conceptualisation are specific to proof systems.

Three classes of error which may occur in interactive proof can be distinguished: syntax, interaction, and logical errors. We make use of the three-level model of interaction to define these error classes according to the interaction level at which the errors occur.

Before describing the error taxonomy in detail, a review of relevant work on errors in programming is presented. This work is particularly relevant to proof assistants such as HOL and Isabelle where proof-as-programming is the dominant interface medium.

2.1. Errors in programming

A classification of programmer errors into syntax, semantic, logical, and clerical errors is proposed by Youngs [30]. Syntax errors are defined to be incorrect expressions of the language in any context. Semantic errors occur when syntactically correct expressions require preconditions which are not met in some specific context. Logical errors are those which cause no malfunction of the program but result in programs that do not work as intended. Clerical errors are syntactic or semantic errors caused by carelessness or accident. No distinction between semantic and logical errors is made in the studies of Sime et al. [31] or of Davis [32], but Davis includes mode errors in his categorisation scheme. Mode errors [33] are commands issued in the wrong context, e.g. to the wrong subsystem of a multi-component application. A classification-based approach to analysing errors is also taken by Eisenstadt and Lewis [34] who propose a list of observable symptoms which occur during interaction with SOLO, a programming environment for novice programmers. Experimental results highlighted aspects of the environment which could be improved. Our study of user errors has similar aims.

A comprehensive taxonomy of user–computer errors is proposed and validated by Zapf et al. [22]. Four classes of errors are identified: functionality problems, usability problems, inefficiency and interaction problems. Functionality problems are the inability to achieve a goal using a particular computer program while usability problems occur when the program is sufficient for the task, but there is a mismatch between user and computer. Usability errors include knowledge errors, memory errors, judgement errors (the failure to understand computer feedback) and thought errors (errors due to a lack of planning). At a lower level of description, usability errors include habit errors (a correct action is performed in the wrong context), omission errors (failure to carry out a well-known action) and recognition errors (when a well-known message is confused with another or ignored). Inefficiency problems may be due to a lack of knowledge or to habit and interaction problems are stated to arise from a lack of co-ordination between individuals in an organisation. For our purposes the most relevant error class is usability errors and we shall make use of the classes which describe errors at the ‘intellectual level of regulation’ [22], i.e. knowledge errors, memory errors and judgement errors.

A rather different view of programmer errors is taken by Davis [35], where a Command Language Grammar [36] description of a task is defined and used to predict errors and mismatches between the task, semantic and syntactic levels of description. This analysis is not simply a categorisation of errors, but an analysis of the task being performed. The issue of mismatched task descriptions has been investigated by Booth [37] who proposes a classification of interaction errors which aims to identify mismatches in the representation of the task between the user and the computer. In this paper, we do not analyse the tasks of the user in such detail. We adopt the taxonomic approach to error analysis.

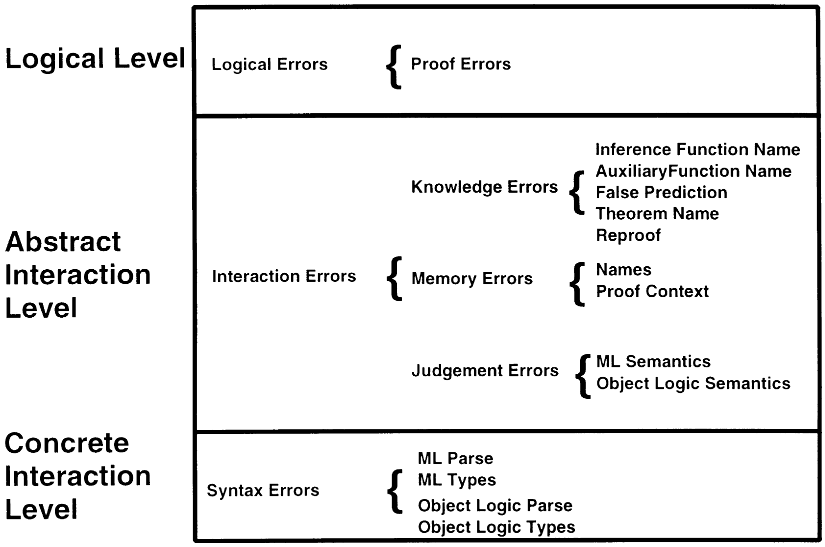

Fig. 2. The error classification.

2.2. Classifying errors in proof

We adopt the conventional definition of syntax errors [30,32]. Syntax errors correspond to errors made at the concrete interaction level.

In tactic-based provers, as in programming tasks in general, there are many means to put a strategy into effect. To distinguish incorrect strategy from incorrect execution of a strategy we use the term *interaction errors*. Interaction errors are distinguished from syntax errors in that the former are identified taking their intended meaning into account, while the meaning of the latter is not relevant. Interaction errors are therefore comparable to semantic errors in the terminology of Youngs [30].[1] The classes of interaction errors we identify are based on the more detailed classification of usability errors of Zapf et al. [22]. Interaction errors are those made at the abstract interaction level of the proposed interaction model.

From the logical perspective, there is often only one natural proof of a theorem. In this case, the proof specifies the ideal strategy. We consider this logical perspective to be comparable to the logical view of program structure in the programming errors literature. Again, this definition follows that of Youngs [30] and in this case we retain the established terminology. Logical errors are those made at the logical level of the interaction model.

The relationship between classes of errors and levels of interaction is shown in Fig. 2. This figure also shows the error taxonomy in detail, and we now give a full description of the classification scheme.

[1] However, we do not wish to the term semantic for this purpose as this would be to overload the use of this term in the logical domain in which we are working.

2.2.1. Syntax errors

The object language/metalanguage distinction is fundamental to tactic-based provers and consequently syntax errors are attributable to two sources: errors in expressing object language formulae and errors in constructing metalanguage expressions. Often, a line of input to the prover will have components in both languages. For example, the HOL command

```
ASSUME_TAC "f (n:num) < = g(n)" ;
```

contains the metalanguage elements `UNDISCH_TAC` (a tactic) and `"…"` (an invocation of the parser function). The item between the quote marks is an object language formula which states that $f(n) \leq g(n)$. In this case, it is necessary to specify the type of the variable n to avoid ambiguity, and so the object-language expression includes the type annotation ‘`:num`’.

Object language formulae must be well-formed and may be annotated with type information, as illustrated. These requirements give rise to two types of object logic error: *object logic parse errors* and *object logic types errors*.

In both HOL and Isabelle, the metalanguage is the programming language ML. The use of ML functions gives rise to a number of possible errors: metalanguage expressions must have the correct syntax and may be typed. For example, the HOL tactic `ASSUME_TAC` requires a term as an argument, hence the object language expression must be converted to a term by the parser—i.e. to have a well-typed command we must have:

tactic “logical expression”.

Consequently, we distinguish *metalanguage types errors* from all other metalanguage errors, which we term *metalanguage parse errors*. Errors due to misspelling words are classed as clerical errors and are not associated with either object or metalanguage errors.

In Isabelle, there is a second metalanguage for the definition of logical types, constants and rules. This theory description language consists of a small number of keywords which structure a text file into various fields-in the style of a mark-up language. This language is untyped. Consequently, *metalanguage types errors* are not possible when defining types, constants and rules in Isabelle. The remaining three classes of error defined above are applicable.

2.2.2. Interaction errors

An object language formula may be syntactically correct, but have a meaning different than intended. This type of error is classed under interaction errors, which we now describe.

One class of interaction errors is attributable to a lack of knowledge on the part of the user about the state of the prover and its commands. This is not to say that the error lies with the user rather than the design of the prover, but that we take an idealised view of what the user needs to know in order to interact correctly. *Knowledge errors* [22] include not knowing the correct name of a tactic, an auxiliary function or a theorem. We term these *inference function name*, *auxiliary function name* and *theorem name errors*, respectively. Auxiliary functions are prover commands that do not perform logical inference. They include commands for navigating the proof tree and commands for modifying the state of the prover. All commands are either inference functions or auxiliary functions.

Knowledge errors also include *false prediction errors*, which occur when the effect of a tactic, or more generally an inference function, does not correspond to the logical strategy being employed. The tactic need not fail for this type of error to occur. If the intended effect of a tactic is in line with the strategy, but the tactic fails because its preconditions are not satisfied, or some other failure occurs, then an error of prediction has occurred because, in reality, the tactic did not match the strategy. Where a tactic does correspond to the logical strategy, but fails because the strategy is incorrect, this is not an interaction error but a logical error.

The final subdivision of knowledge errors is *reproof errors*. An error of this class occurs when the user proves a theorem whose truth has already been established and can be found in one of the theorem-prover's libraries.

Exploring sequences of tactics which might lead to a proof is an integral part of proof activity. We do not consider unsuccessfully exploring part of the search space to constitute an interaction error in itself. But an unsuccessful sequence of steps may end with an interaction error, for example, the failure of a tactic, or with the realisation that the proof strategy is incorrect.

Interaction errors may also be due to forgetting a piece of information generated during the proof. These *memory errors* [22] include forgetting the current proof context, as may occur after several backing up steps, and forgetting a previously made definition or binding. We term these *proof context errors* and *names errors*, respectively.

The third class of interaction errors is *judgement errors* which are defined as errors in understanding the feedback from the computer [22]. We broaden this definition to include any misinterpretation of information represented on the computer. This class has two subclasses: *object language semantics errors*, which are errors in interpreting the meaning of object language expressions, and *metalanguage semantics errors*, which are errors in interpreting the meaning of metalanguage expressions. These semantic errors include the misinterpretation of logical connectives and also any error in understanding the status of symbols, i.e. as constants, variables or meta-variables. In the metalanguage, these errors include any incorrect association of function to command name. As we cannot objectively determine the meaning of ill-formed expressions and, therefore, cannot determine whether this is the desired meaning, judgement errors can be made only with well-formed expressions.

2.2.3. Logical errors

The third main category of errors is logical errors. There is only one subclass: proof errors. An error of this type occurs when a tactic which implements an identifiably incorrect proof strategy is executed. For example, the user may stipulate a case-analysis where an induction is logically required.

3. Evaluation

The proposed taxonomy distinguishes the classes of errors that we believe to be important in the theorem-proving domain. The classes were defined analytically—by considering features of the notation and the interactive environment. In order for the error taxonomy to be a useful usability metric it must categorise the errors that actually occur in real interactions. That is, its coverage must be evaluated empirically. A second factor in the utility of this approach is whether the error categories capture features of the interaction which vary across user groups or across systems. Variations in error rates should have implications for design as improving usability is the ultimate goal of our research. In order to validate the taxonomic approach to characterising usability two experimental trials were carried out. The experimental method is described below, and the results are presented in Section 4.

3.1. Errors during proof

The error types listed in the error taxonomy are identified and quantified as follows:

Syntax errors: Errors in object or metalanguage expressions may occur whenever a line of input to the prover contains either type of expression. Whether a line of input contains an object or a metalanguage element is easily determined, as is the cause of the error. Syntax errors are expressed as the ratio of the number of lines containing an error to the number of lines containing a language element of that type.

Knowledge errors: Errors in the use of inference function names are expressed as the ratio of the number of incorrect names entered at the command line to the total number of correct inference function names used in the proof attempt. This requires incorrect inference function names to be identified—whether a line of input is intended to be tactic is usually obvious from the context. Errors in the use of auxiliary function names and theorem names are quantified in a similar way.

Errors in predicting the effects of a proof step are expressed as a ratio of the number of steps which clearly had an unintended outcome (usually failure) to the total number of proof steps. The number of reproof errors is the sum of the number of lemmas which exist in the prover’s library but are proven again by the subject during the trial.

Memory errors: Proof context errors are actions executed at the wrong place in the proof tree. They are expressed as a ratio of the number of times an action is executed in the wrong context to the number of times the proof state changes. Names errors are expressed as the ratio of incorrectly recalled user-defined names to the total number of user-defined names.

Judgement errors: Errors in the interpretation of well-formed object language expressions are possible whenever such an expression is entered at the command line. The intended meaning of an expression from the subject’s viewpoint must be inferred from the expression itself and the way in which the expression is used in the proof. Where the actual meaning does not correspond to that intended by the subject an error occurs. For example, if a subject incorrectly formulates the pattern of quantifiers in the goal expression, and sets this expression as the goal, then an error of interpretation has taken place as we can objectively determine the real meaning of the goal expression, and can compare this to the goal as specified in the problem statement. Semantics errors are expressed as a ratio of the number of incorrectly interpreted expressions to the total number well-formed expressions.

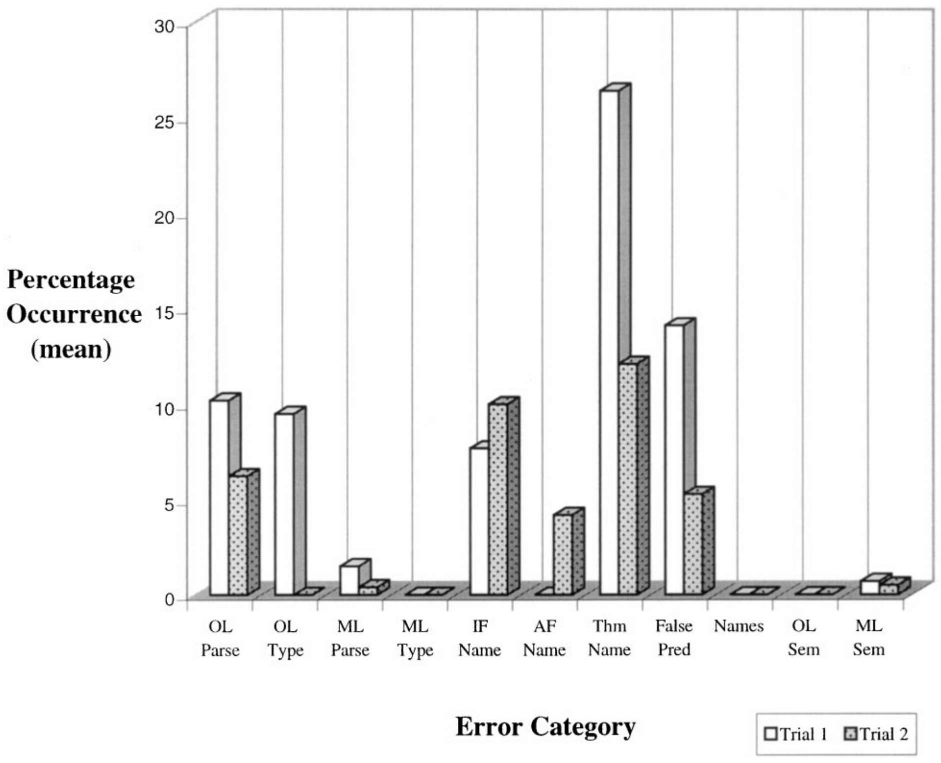

Fig. 3. Error data from trials 1 and 2.

3.2. Errors in theory definition

Only syntax errors are possible in the theory definition task. In Isabelle, theory definition involves editing a file and loading it. The only command involved is the `use_thy` command and errors at the abstract interaction level (i.e. errors in the use of this single command) are not interesting. Errors in theory definition are quantified by counting the errors corrected and expressing this as a fraction of the number of times a theory file is changed in any way.

3.3. Documenting error

It has been possible to define in advance of data analysis how specific types of error are identified and counted. However, it is also necessary to identify errors which fall outside these categories. This is done by annotating the logs of user interaction in all places where interaction is inhibited, e.g. where the prover rejects the users input as invalid, or where the input is valid but has an unintended effect. An explanation for the cause of the error is proposed, and if the taxonomic definitions do not capture the error, or its cause, then the error will be said to lie outside the taxonomy. In future, the taxonomy might be extended to cover new types of error, should any be identified.

3.4. Experimental method

The aims of the empirical work are to obtain some basic quantitative data on user performance and to evaluate the error taxonomy as a usability tool. A further aim is to be able to draw some conclusions about the usability of interactive theorem provers, assuming that measuring errors proves to be a valid assessment technique.

The hypothesis of the evaluation is that the error taxonomy adequately characterises user input errors (within the three level framework described above). If a significant number of user errors cannot be classified, then this hypothesis will be rejected. We argue that observation of the behaviour of a group of users with comparable experience provides sufficient evidence for this evaluation. This approach is open to criticism as variables such as the opportunity for making errors are not controlled. To meet this criticism, the number of opportunities for error must be established for each error type. Where the number of opportunities for error is insignificant then the evaluation will be inconclusive. The evaluation requires the collection and analysis of user data. The trials were carried out in an environment which was similar to the subject's normal working environment. They were conducted in a uniform manner: the trial procedure was described to the subject, information regarding the subject's experience was collected, and finally the subject was presented with the theorem to be proved.

The collection of experimental data might have allowed more specific hypotheses about user performance to be tested. However, there are no reported quantitative studies of errors in theorem proving, and hence there is little material on which to base a hypothesis about the likelihood of user errors of different types. There are differences between HOL and Isabelle in the way in which the syntax of the object logic must be specified. It is to be hoped that the error taxonomy approach will highlight different error rates in object logic syntax, however, no established sources can be drawn upon to justify this claim as a hypothesis. There are no widely used alternatives to the editor/command line interface to HOL or Isabelle. Consequently, it was not possible to perform a comparative evaluation of different interfaces. By collecting data on the usability of the current standard interface a baseline for future comparative studies is established. Experimental design is further restricted by the long learning curve associated with theorem provers. The acquired skills are also system specific. It was not possible to train subjects to be experts in both HOL and Isabelle, neither was it possible to find subjects who were experts in both systems. As a result, control of the training of subjects, and control of the subject group were ruled out.

4. Results

Two trials were carried out, one using the HOL theorem prover, the other using Isabelle. In both cases, subjects had a high level of experience. The results of the trials are now presented and we note where observed errors are not captured by the taxonomy. Section 5 analyses the results, highlighting the similarities and differences between the trials, and summarising our conclusions on the coverage of the taxonomy.

4.1. Results of trial 1

All subjects were given the task of proving the following list induction theorem using the HOL prover:

$$\vdash \forall P \cdot (P[\,] \wedge (\forall x \cdot P[x] \wedge \forall l_1 l_2 \cdot (P l_1 \wedge P l_2) \supset P(\mathsf{APPEND} l_1 l_2)) \supset \forall l \cdot P l$$

All subjects were frequent or very frequent users and had a wide range of total experience with HOL—from 7 to 120 months (all results are tabulated in Appendix A). The total number of interactions with HOL ranged from 13 to 50. The results of this trial are presented in Fig. 3 which shows the mean percentage error rate across all trials. A full listing of each of the six proof sessions, and their analysis, can be found in [38].

Object language errors occur more frequently than metalanguage errors, by a factor of 13 in terms of percentage occurrence. Within object language errors, parse and types errors occur with equal frequency of approximately 10%. The only metalanguage syntax errors observed were parse errors.

Knowledge errors constitute the largest class of errors. Theorem name errors are twice as frequent as false prediction errors, which in turn are twice as frequent as inference function name errors. The false prediction error rate is statistically significant (at the 5% point of the t-distribution) even for the small sample size of this trial.

Three subjects reproved a theorem, namely `CONS_APPEND`, which they could have found in a HOL library, while only one subject (5) found the theorem and used it. Two subjects (1 and 2) avoided using the `CONS_APPEND` theorem directly by making their proofs more convoluted. As these subjects did not prove the theorem explicitly the tabulated value in Appendix A is 0, according to the definition of reproof errors given earlier.

In absolute terms, two context errors are made by the six subjects, sample mean 1.9%. Across the six trials, ten names are defined and these give rise to no memory errors. Subjects make no judgement errors about object language semantics, and two errors about metalanguage semantics.

All subjects decided that the problem could be solved by list induction. One logical error was made by subject 5 who attempted to solve the step case by list induction. This attempt was abandoned after one proof step.

All observed errors were categorised by the taxonomy.

4.2. Results of trial 2

Trial 2 differs from trial 1 in that subjects used the Isabelle prover, instantiated with the HOL object logic. In addition, the scope of trial 2 was expanded to include a preliminary theory definition phase. Theory definition is an important task in theorem proving, but supporting users in this task in interface design is often neglected in favour of supporting the proof task.

Subjects were required to formulate the following definition prior to starting the proof, given the informal specification `funpow` $f\ n\ x = f^n(x)$:

$$\begin{array}{llll} \mathsf{funpow} & f & 0 & x = x \\ \mathsf{funpow} & f & (n+1) & x = f\ (\mathsf{funpow}\ f\ n\ x) \end{array}$$

Table 1
Theory definition

Subject	Theories	Changes	Corrections	Time (min:s)
A	1	0	0	2:32
B	4	3	3	6:09
C	4	3	2	10:31
D	7	6	4	13:50
E	20	20	9	30:44
F	6	6	4	13:35
Mean	7.0	6.3	3.7	

They were then asked to prove the following theorem:

$$\vdash \forall x\ f\ n\ m \cdot \mathsf{funpow}\ f(n+m)\ x = (\mathsf{funpow}\ f\ n)(\mathsf{funpow}\ f\ mx)$$

The subjects' experience with the prover ranged from 6 to 96 months, and the number of interactions required to complete the task ranged from 10 to 45. Appendix A presents these data in detail. We consider theory definition and proof as distinct activities and now consider the errors occurring in each.

Data on the number of theories which subjects specified, up to and including the final (correct) theory, is given in Table 1. The data indicate that the task of making a definition is not straightforward, requiring an average of 7 iterations. The average is reduced to 4.4 if subject E, who had particular difficulty, is excluded from the calculation, but is still considerable. An average of 41% of the changes made during this task are incorrect (28% if subject E is excluded from the calculation); that is, they introduce a new error or do not fix an existing error. The times quoted are the times up to the setting of the goal, but this does not preclude further changes to the theory file later in the trial. The mean time taken to formulate a syntactically correct theory, as a percentage of the total time for the exercise, is 28%.

Errors occurring during theory definition are shown in Table 2. Errors of specific types are calculated with respect to the number of corrections (as given in Table 1) as this corresponds to the number of theory loading actions.[2] Object language errors are approximately twice as frequent as metalanguage errors. Errors concerning object language types are the more frequent error of this class. It should be noted that the metalanguage used in theory definition is one which delimits the various types of definitions (types, constants, and rules) and is not typed. Subjects B, D, E and F also made errors concerning the library theories which they included in their own theory. This class of error cannot be categorised in our scheme and for this reason the tabulated number of errors does not add up to the total number of corrections in these cases. We postulate a new class of error, context errors, to account for these slips.

Errors occurring during the proof attempts are given in Fig. 3. The results show that object logic parse errors are the only source of syntax error. Knowledge errors constitute

[2] As a correction is only required when an error occurs, the ratio of errors to actions is not comparable with the ratio for proof errors as actions during proof are not necessarily errors, while the action of correcting a theory file necessarily results from an error.

Table 2
Syntax errors in theory definition trial 2

Subject	OL parse errors	OL types errors	ML parse errors	ML types errors
A	–	–	–	–
B	0/3	1/3	1/3	–
C	1/2	0/2	1/2	–
D	0/4	3/4	0/4	–
E	5/9	3/9	0/9	–
F	0/4	1/4	2/4	–
Mean (%)	21.1	33.3	26.7	–

the majority of errors. Within this class, theorem name errors and inference function name errors are equally significant, and are approximately twice as frequent as auxiliary function names and false prediction errors. One reproof error occurs when subject F proves a lemma unnecessarily. The lemma in question is in the `arith_ss` simplification set, but subject F is unable to find it and, after misdiagnosing a proof error, reproves the lemma.

No significant error rates are measured for memory or judgement errors. A total of six logical errors are made by the six subjects. These involve failing to eliminate the leading quantifiers of the goal (three subjects) and doing induction on the wrong variable (two subjects).

All proof errors were categorised by the taxonomy, 19% of theory definition errors lay outside the taxonomy.

5. Analysis

The coverage of observed errors by the taxonomy is adequate: 76 of the 80 observed errors can be classified. Consequently, we can justify examining the categorised data in more detail (with the qualification that the small sample size obviously limits the certainty of the claims we can make). We begin by comparing the complexity of the tasks in the two trials.

The tasks are of similar complexity: a mean of 36 interactions are recorded in trial 1, 33 in trial 2. The average proof requires a mean of 12.8 and 13.8 proof steps respectively, and the times spent on the proof task are also broadly comparable (mean times to complete the proofs are 18:40 and 25:29 min).

We have proposed a proof-as-programming view of interactive theorem proving and argued that programming is the medium of the interaction. The empirical study and analysis presented here allows us to ask whether the manipulation of the proof state using commands is error prone, and whether the environment which supports this interaction is itself a source of error. We can also examine the complexity of constructing a proof in each system.

Errors in the recall and application of inference functions are captured by inference function name and false prediction errors. The false prediction error rate of 14.1% found in trial 1 is not high in absolute terms, but is amongst the most significant rate observed. However, this rate is reduced to 5.3% in trial 2, indicating that the implementation of the state manipulation function is an important factor in the usability of the programming medium. Errors in recalling function names are similar in the two trials, approximately 10%.

A high error rate in the use of auxiliary functions and in the construction of metalanguage expressions would indicate problems with the programming environment. Our results show that fewer errors occur in the use of auxiliary functions than in the use of inference functions and that metalanguage expressions are a lesser source of error than object logic expressions. Consequently, we conclude that the environment is not a significant source of error.

The number of inference functions and auxiliary functions must be taken into account when looking at the complexity of constructing a proof. HOL proofs contain a greater number of inference functions, a mean of 15, in comparison with the typical Isabelle proof where 5.4 different inference functions are used. The mean number of auxiliary functions used is 6.7 and 7.8, respectively. These figures show that the typical Isabelle proof requires more auxiliary functions than tactics, while the reverse is true of the average HOL proof. By carefully classifying errors, we can see that although Isabelle users need to know fewer tactics than HOL users, they need to be able to use other system functions, and the trial data indicates that errors are not necessarily reduced as a result.

The observation of the theory definition phase in trial 2 allows us to conclude that a significant number of syntax errors occur in this task. We can also begin to determine the impact of this task on the subsequent proof task. The frequency of syntax errors during the proof phase of trial 2 is lower than that of trial 1. This is due in part to the definition of types in the Isabelle theory file, which means that users do not need to annotate the goal expressions with type information—types are a notable source of error in trial 1. This indicates that while syntax errors are moved into the earlier formalisation phase and are not eliminated, there may be an advantage in this approach as interruptions to the interactive proof phase are reduced.

Low error rates for experienced users do not indicate that the command line environment is without problems. Error messages are often difficult to interpret and help provision is minimal. Improvements to the environment are now discussed.

6. Conclusions

The proposed error taxonomy has been shown to capture the majority of errors observable in interactive proof attempts. It can therefore be considered adequate as a usability metric. The results of trials on two different theorem provers indicate that error analysis can identify differences in error distribution between the classes of syntax errors and interaction errors.

In the systems selected for these trials, the interface was simply the command line of the theorem prover and the differences in user error are explainable by considering the underlying theorem-proving technology. User interaction will be significantly modified by the addition of graphical components into theorem-prover interfaces, and an analysis of errors can be expected to reveal how graphical components assist or hinder user interaction, in comparison with text-based interfaces. In this case, the explanation of differences in error frequency will be in terms of interface features, rather than in terms of theorem-proving technology.

For example, if tactic construction was to be aided by selecting the inference function from a menu it might be expected that inference function name errors and metalanguage parse errors would decrease. An increase in metalanguage semantics errors could also occur as users might select functions they did not understand. An experimental evaluation would be required to determine the advantages of the menu feature. A more radical redesign of the interface where some tactics are replaced by direct manipulation would eliminate several classes of error as the user would not be required to construct the tactic. However, new types of concrete interaction error would be possible. The user would also have to switch between different models of how to translate from abstract proof to concrete interaction and logical errors might increase.

The results of the trials reported here have a number of practical implications for the design of proof environments and the user interface:

- Improved support for the entry of object logic expressions and improved feedback of error information would be valuable.
- The introduction of a distinct theory definition phase may reduce proof phase errors. In addition, adequate interactive feedback of errors in the theory is required.
- Better theorem search and matching tools could reduce two sources of error (theorem name and reproof errors).
- Improved access to information about defined constants, theory structure, tactic names and specification would aid users who cannot recall this information from memory, but can readily make use of it once it has been found.

The evaluation method we have presented allows us to distinguish the classes of error that are of most interest to us from a usability perspective. The taxonomy is derived from an analysis of the notations and programming environment as presented to the user, rather than from a general analysis of the causes of human error (see [33] for such a treatment). Consequently, a taxonomy which is constructed by this method will always be relevant to a specific class of interactive systems, rather than to all systems. We consider this concreteness to be a positive feature. Our approach requires us to make only minimal assumptions about the problem solving activity of subjects. Therefore, subjects can be given realistic tasks to perform and can be observed in their typical working environment. These are important features of a practical methodology for usability assessment.

Appendix A

The subjects' experience with the prover and the number of interactions required to complete the task are detailed in Tables A.1–A.6.

Table A.1
Basic parameters trial 1 (abbreviations: VF, very frequent; F, frequent)

Subject	HOL experience (months)	Current HOL usage	Number of interactions	Time (min:s)
1	54	VF	13	7:36
2	12	VF	38	10:44
3	7	VF	50	19:25
4	120	F	32	10:32
5	108	F	39	33:15
6	96	F	44	30:32

Table A.2
Syntax errors trial 1

Subject	OL parse errors	OL types errors	ML parse errors	ML types errors
1	0/3	0/3	0/13	0/13
2	0/10	1/10	1/35	0/35
3	1/9	2/9	0/46	0/46
4	0/6	0/6	1/28	0/28
5	3/6	0/6	1/39	0/39
6	0/16	4/16	0/42	0/42
Mean (%)	10.2	9.5	1.5	0

Table A.3
Interaction errors trial 1

Subject	Knowledge errors					Memory errors		Judgement errors	
	IF name	AF name	Theorem name	False prediction	Re-proof	Proof context	Names	OL semantics	ML semantics
1	0/13	0/6	0/1	0/6	0	0/7	–	0/3	0/13
2	1/14	0/5	0/2	1/14	0	2/18	0/1	0/10	0/35
3	2/14	0/9	1/3	3/14	1	0/19	0/1	0/9	2/46
4	0/16	0/6	0/3	2/12	1	0/10	0/2	0/6	0/28
5	2/16	0/8	3/4	1/16	0	0/16	0/2	0/6	0/39
6	0/17	0/6	1/2	5/15	1	0/17	0/4	0/16	0/42
Mean (%)	7.7	0	26.4	14.1	–	1.9	0	0	0.7

Table A.4
Basic parameters (abbreviations: VF, very frequent; F, frequent)

Subject	Experience (months)	Current usage	Number of interactions	Total time (min:s)	Proof time (min:s)
A	96	VF	10	8:05	5:33
B	6	VF	26	38:24	32:15
C	30	F	45	47:24	36:53
D	30	F	42	32:21	18:31
E	18	F	37	41:26	10:42
F	6	F	38	47:48	34:13

Table A.5
Syntax errors in proof Trial 2

Subject	OL parse errors	OL types errors	ML parse errors	ML types errors
A	0/3	0/3	0/10	0/10
B	0/7	0/7	0/26	0/26
C	0/11	0/11	0/45	0/45
D	3/8	0/8	1/42	0/42
E[a]	0/1	0/1	0/37	0/37
F	0/13	0/13	0/38	0/38
Mean (%)	6.25	0	0.4	0

[a] This subject did not complete the proof.

Table A.6
Interaction errors trial 2

Subject	Knowledge errors					Memory errors		Judgement errors	
	IF name	AF name	Theorem name	False prediction	Re-proof	Proof context	Names	OL semantics	ML semantics
A	0/4	0/5	0/2	0/4	0	0/5	0/3	0/3	0/10
B	1/5	0/6	0/4	0/17	0	0/18	0/3	0/7	0/26
C	0/8	0/10	1/7	2/21	0	0/24	0/4	0/11	0/45
D	1/5	0/11	1/3	2/9	0	0/11	0/5	0/8	0/42
E[a]	0/1	1/4	1/4	0/9	0	0/10	0/1	0/1	1/37
F	1/5	0/7	0/5	0/18	1	0/19	0/4	0/13	0/38
Mean (%)	10.0	4.2	12.1	5.3	-	0	0	0	0.5

[a] This subject did not complete the proof.

References

[1] E.M. Clarke, J.M. Wing, Formal methods: state of the art and future directions, ACM Computing Surveys 28 (4) (1996) 626–643.

[2] L.C. Paulson, Isabelle: a generic theorem prover, Lecture Notes in Computer Science, 828, Springer, Berlin, 1994.

[3] M.J.C. Gordon, T.F. Melham (Eds.), Introduction to HOL: a Theorem Proving Environment for Higher Order Logic Cambridge University Press, Cambridge, UK, 1993.

[4] S. Owre, J.M. Rushby, N. Shankar, PVS: a prototype verification system, in: D. Kapur (Ed.), Proceedings of the 11th Conference on Automated Deduction, LNAI, 607, Springer, Berlin, 1992, pp. 748–752.

[5] S.P. Miller, M. Srivas, Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods, Workshop on Industrial Strength Formal Methods, WIFT'95, IEEE Computer Society Press, Silver Spring, MD, 1995.

[6] A.J. Camilleri, A hybrid approach to verifying liveness in a symmetric multi-processor, Proceedings of the 10th International Conference on Theorem Proving in Higher-Order Logics, LNAI, 1275, Springer, Berlin, 1997 pp. 49–67.

[7] E.L. Gunter, Adding external decision procedures to HOL securely, Proceedings of the 11th International Conference on Theorem Proving in Higher-Order Logics, LNAI, 1479, Springer, Berlin, 1998 pp. 143–152.

[8] A. Cant, M.A. Ozols, Xisabelle: a graphical user interface to the Isabelle theorem prover, Information Technology Division, DSTO, P.O. Box 1500, Salisbury, South Australia 5108, 1995.
[9] L. Théry, A proof development system for the HOL theorem prover, in: J.J. Joyce, C.-J.H. Seger (Eds.), Higher Order Logic Theorem Proving and its Applications, Lecture Notes in Computer Science, 780, Springer, Berlin, 1994, pp. 115–128.
[10] D. Syme, A new interface for HOL: ideas, issues, and implementation, in: E.T. Schubert, P.J. Windley, J. Alves-Foss (Eds.), Higher Order Logic Theorem Proving and its Applications, Lecture Notes in Computer Science, 971, Springer, Berlin, 1995, pp. 324–339.
[11] J. Bertot, Y. Bertot, The CtCoq experience, in: N. Merriam (Ed.), User Interfaces for Theorem Provers: UITP'96, University of York, York, UK, 1996 Technical Report.
[12] Y. Bertot, L. Théry, A generic approach to building user interfaces for theorem provers, Journal of Symbolic Computation 25 (1998) 161–194.
[13] R. Bornat, B. Sufrin, Jape's quiet interface, in: N. Merriam (Ed.), User Interfaces for Theorem Provers: UITP'96, University of York, York, UK, 1996 Technical Report.
[14] Y. Bertot, T. Kleymann, D. Sequeira, Implementing proof by pointing without a structure editor, LFCS Report Series ECS-LFCS-97-368, University of Edinburgh, Edinburgh, UK, 1997.
[15] D.A. Norman, The Psychology of Everyday Things, Basic Books, New York, 1988.
[16] N.A. Merriam, A.M. Dearden, M.D. Harrison, Assessing theorem proving assistants: concepts and criteria, Proceedings of the Workshop on User Interface Design for Theorem Proving Systems, Department of Computing Science, Glasgow, 1995
[17] J. Rasmussen, A. Pejtersen, K. Schmidt, Taxonomy for cognitive work analysis, Technical Report M-2871, Riso National Laboratory, Roskilde, Denmark, September 1990.
[18] M. Eisenstadt, M. Brayshaw, The transparent Prolog machine (TPM): an execution model and graphical debugger for logic programming, Journal of Logic Programming 5 (1988) 277–342.
[19] M. Brayshaw, M. Eisenstadt, Adding data and procedure abstraction to the Transparent Prolog Machine (TPM), in: K. Bowen, R. Kowalski (Eds.), Proceedings of the Fifth International Conference Symposium on Logic Programming, MIT Press, Cambridge, MA, 1988.
[20] H. Lieberman, C. Fry, Bridging the gulf between code and behaviour in programming, Proceedings of ACM Conference of Human Factors and Computing Systems (CHI) Denver, 1995, 480–486.
[21] P. Mullholland, A principled approach to the evaluation of software visualization: a case-study in Prolog, in: J. Stasko, J. Domingue, M. Brown, B. Price (Eds.), Software Visualization: Programming as a Mutli-media Experience, MIT Press, Cambridge, MA, 1998, 439–452.
[22] D. Zapf, F.C. Broadbeck, M. Frese, H. Peters, J. Prumper, Errors in working with office computers: a first validation of a taxonomy for observed errors in a field setting, International Journal of Human–Computer Interaction 4 (4) (1992) 311–339.
[23] H. Lowe, D. Duncan, XBarnacle: making theorem provers more accessible, Proceedings of the Fourteenth Conference on Automated Deduction, Townsville, Australia, LNAI, 1249, Springer, Berlin, 1997 pp. 404–407.
[24] T. Schubert, A tree-based, graphical interface for large proof development, in: T.F. Melham, J. Camilleri (Eds.), Supplementary Proceedings of the Seventh International Workshop on Higher Order Logic Theorem Proving and its Applications, University of Malta, Valletta, September 1994, , 1994.
[25] S.J. Garland, J.V. Guttag, An overview of LP: the Larch Prover, in: N. Dershowitz (Ed.), Proceeding of the Third Conference on Rewriting Techniques and Applications, Springer, Berlin, 1993, 137–151.
[26] M.J.C. Gordon, R. Milner, C.P. Wadsworth, Edinburgh LCF: a Mechanised Logic of Computation, LCNS, 78, Springer, Berlin, 1979.
[27] J. Neilsen, A virtual protocol model for human–computer interaction, International Journal of Man–Machine Studies 24 (1986) 31–312.
[28] E. Hutchins, J. Holland, D. Norman, Direct manipulation interfaces, in: D. Norman, S. Draper (Eds.), User Centred System Design, Erlbaum, Hillsdale, NJ, 1986.
[29] J.S. Aitken, P. Gray, T. Melham, M. Thomas, Interactive theorem proving: a study of user activity, Journal of Symbolic Computation 25 (1998) 263–284.
[30] E.A. Youngs, Human errors in programming, International Journal of Man–Machine Studies 6 (3) (1974) 361–376.
[31] M.E. Sime, A.T. Arblaster, T.R.G. Green, Reducing programming errors in nested conditionals by

prescribing a writing procedure, International Journal of Man–Machine Studies 9 (1) (1977) 119–126.

[32] R. Davis, User error or computer error? Observations on a statistics package, International Journal of Man–Machine Studies 19 (4) (1983) 359–376.

[33] D.A. Norman, Categorisation of action slips, Psychological Review 88 (1981) 1–15.

[34] M. Eisenstadt, M.W. Lewis, Errors in an interactive programming environment: causes and cures in novice programming environments, in: M. Eisenstadt, M. Keane, T. Rajan (Eds.), Explorations in Human–-Computer Interaction and Artificial Intelligence, Lawrence Erlbaum Associates, Hillsdale, NJ, 1992.

[35] R. Davis, Task analysis and user errors: a methodology for assessing interactions, International Journal of Man–Machine Studies 19 (6) (1983) 561–574.

[36] T.P. Moran, The command language grammar: a representation for the user interface of interactive computer systems, International Journal of Man–Machine Studies 15 (1) (1981) 3–50.

[37] P.A. Booth, Identifying and interpreting design errors, International Journal of Human–Computer Interaction 2 (4) (1990) 307–332.

[38] S. Aitken, P. Gray, T. Melham, M. Thomas, ITP Project Anthology, Technical Report TR–1997–36, Department of Computing Science, University of Glasgow November, 1997.