Project Report
Theorem Prover Usability

James Cheney
COMM 641

May 8, 2001

1 Introduction

There is a large research group in the Cornell CS department on automated reasoning and theorem proving. They have developed a system of formal mathematics and a program called Nuprl[5] that allows users to state and prove theorems about mathematical and computational systems in a fully verifiable way. The vision for systems of this kind is that researchers and programmers ought to be able to rely on formal mathematical tools like Nuprl to do their jobs better, because these tools can help analyze problems, solve easy problems, and can check whether a proposed solution does in fact solve a problem, or if there is a gap in the argument. Recently the Nuprl group has been awarded a grant for the purpose of building online digital libraries of formal mathematics; in a sense, this is complementary to building an "online community" of people interested in communicating in a medium of formal mathematics. It is hoped that in the presence of the right kind of tools, a "open mathematics" community analogous to the "open source" community might grow and help fuel the vision for mathematical tools.

However, the Nuprl system has a very complex user interface and is only really usable by people in this research group willing to devote a lot of effort to the task. The original versions of the system were command-line based. More recent versions are "graphical" in the sense that they consist of multiple windows of text that uses a special font containing mathematical symbols; however, the mode of operation is still mostly command-line, but with several obscure key and mouse commands thrown in. In these systems, the interface is solidly coupled to the underlying system, making it difficult to imagine replacing the interface without writing a whole new system. The most recent version of Nuprl[4] has moved towards a more modular design, separating the "editor", "library", and "refiner" components. Thus it is now possible to imagine ripping out the editor (the chief user interface component) and replacing it with a usable one.

In this project we analyzed the existing Nuprl editor user interface, surveyed users' views of the interface, and built prototypes for redesigning and improving it based on the survey results.

2 The Current Nuprl Interface

There exist currently two versions of Nuprl in use, Nuprl 4.2 and Nuprl 5.0. Although they have somewhat different interfaces, they are otherwise similar in conceptual design and organization. We have only been able to work with Nuprl 4.2, so we will describe that interface.

2.1 High-level description

Nuprl is an *interactive theorem prover* (or *proof assistant*). It provides a formal language for writing down logical and mathematical statements, for example:

$$P(x) \wedge (\forall y.P(y) \rightarrow Q(y)) \rightarrow Q(x)$$

which states that for properties P and Q, that if $P(x)$ is true and whenever $P(y)$ is true, so is $Q(y)$, then $Q(x)$ must be true. P and Q are *propositional functions* which might stand for a larger formula involving an argument x. For example, $P(x)$ might be "$x \geq 1$" and $Q(x)$ might be "$x \geq 0$" (these interpretations of P and Q would make the second condition true, since whenever a number is greater than 1, it is also greater than 0).

Expressions (or *terms*) such as $x+1$, and formulas (or *propositions*) such as $x > 5$, are input into the system using a *structured editor*. The standard choice for entering symbolic data of this kind into computers has historically been through *parsing*, in which the program reads a string of symbols `x + 1` and constructs a *parse tree* $plus(variable(x), number(1))$ for the expression. This parsing process is sensitive to mistakes on the part of the user, called *syntax errors*. In contrast, terms are entered into Nuprl using a *structured editor*. In structured editing, the user can manipulate the parse tree of the term directly, rather than indirectly as a sequence of characters. As a result, only syntactically correct expressions can be formed, so (in principle) there are no syntax errors.

Terms are stored internally as parse trees such as $plus(variable(x), number(1))$; however, this notation is much less readable than the customary $x + 1$. To remedy this deficiency, Nuprl has a mechanism called *display forms* that allows terms like $plus(variable(x), number(1))$ to be visualized in more intuitive notations $x+1$. This mechanism is extensible so that the user can introduce new notations.

In addition to these facilities for *expressing* and *displaying* formulas, Nuprl provides another facility for *verifying* or *proving* them. This is the principal task for which Nuprl has been developed; the term language, structured editor, and display forms all support this goal. In Nuprl, proofs are also tree-structured, with the proof goal (theorem) at the root and subgoals at the other nodes. The tree nodes are also labeled with *proof steps*, which can have many forms. The most basic form of proof step is an elementary rule, such as the "and-introduction" rule, which states: "If the goal is to prove $A \wedge B$, then the subgoals are to prove A and to prove B."

Proving theorems using only basic rules can be tiresome, and so Nuprl provides some automation abilities. It is possible to write small programs in Nuprl called *tactics* that attempt to apply basic rules (or other tactics) to the current goal. The most common examples in Nuprl are the **D** and **Auto** tactics. The **D** tactic decomposes the outermost logical connective of a formula using the appropriate basic rule; for example applying **D** to $A \wedge B$ would result in using the and-introduction rule. The **Auto** tactic repeatedly applies several (complicated) tactics to a goal until no progress is made. Higher-order tactics, or *tacticals*, allow basic tactics to be combined in useful ways. For example, the **THEN** tactical takes two tactic arguments, and tries the first tactic and then the second (unless the first fails). Finally, there are special tactics, called *decision procedures* and *rewrites*. Decision procedures attempt to determine the truth or falsity of an equation (such as an arithmetic inequality) exactly; rewrites simplify equations or terms using mathematical rules.

Nuprl uses a special form of mathematics called *constructive* mathematics, which means that proofs have computational content (programs) associated with them. Thus, proving a theorem that says that a certain equation has a solution involves writing a program that solves the equation. Nuprl makes it possible for users to extract these programs from proofs. Theoretically, at least, one could imagine formalizing a complex problem in Nuprl, proving that a solution exists, and then extracting the solution as a program that solves the problem. In reality, this can be very difficult.

Verification tasks often rely on many theorems, definitions and proofs; thus, Nuprl provides a *library* facility for storing and collecting definitions, theorems and proofs, and other relevant information.

2.2 Low-level description

The current versions of Nuprl are implemented on Unix systems using the X Windows graphical interface. Older versions of Nuprl were command-line only. The current version is still tied to the command line (see Fig. 1), but the windowed interface makes it possible to display several different kinds of information at once and also to select certain actions and information with the mouse. Nevertheless, Nuprl's graphical interface lacks many standard GUI features, such as dialogs, menus, toolbars, and so on. Most interaction consists of selecting the appropriate window with a mouse click, then entering data using keystrokes.

The structured editor (see Fig. 2) works principally through command keystrokes. Special keys are used to insert a new subterm, delete subterms, and copy and paste. The mouse can be used to select subterms also. The term's display forms can also be activated ("imploding" the term) or deactivated ("exploding" it) during editing. Customized display forms can be created as library objects using a specialized display form language.

Proofs are displayed in a separate window (see Fig. 3). A proof window displays the hypotheses and main goal at the top, then a space for the proof rule, then a list of subgoals generated by the rule. The proof tree can be navigated using the cursor to move up to the parent node (the goal) or down to the child nodes (the subgoals). Proof steps (that is, tactics) can be entered by clicking on the space for the tactic and entering tactic text in a pop-up window. Tactics themselves are programs, so they might consist of a single existing tactic or a tactic-producing function applied to several arguments. For example, the **Auto** tactic is stand-alone, whereas **D** is a function that takes a number indicating which part of the goal to decompose (0 for the main goal, or $1, 2, 3, \ldots$ for hypotheses). Similarly, the **THEN** tactical takes two tactics A, B as arguments, which is expressed A **THEN** B.

Theorems and proofs, display forms, definitions, and code extracted from proofs are different classes of *library objects*. Library objects are listed by name in a library window (bottom half, Fig. 1). These objects can be created, deleted, opened for editing, and loaded or saved using commands entered into Nuprl's interaction loop, also displayed in a window. Additional user-defined commands (such as tactics) can be entered as code into files that Nuprl loads when started.

3 Survey Design

Our intuition is that the Nuprl interface is unnecessarily hard to use, and could be improved. The authorfound Nuprl difficult to use, despite being motivated to learn how to use tools like Nuprl and despite having a technical background that should be an advantage in understanding such tools. Consequently, we feel that it would be worthwhile to redesign the interface. We designed a survey to measure users' perceptions of Nuprl's interface. The intent of this survey was to determine what aspects of Nuprl were both important (frequently encountered) and difficult to perform. Since we got responses from both novices and experienced users/developers of the system, we hoped to also be able to measure which tasks were most difficult to learn. Reliable information about these problem features would be helpful in deciding where to concentrate efforts in improving the interface.

3.1 Questions

The survey (see appendix) consisted of a background part, a qualitative (free-response) part, and a quantitative part. The background questions asked how long the user had used Nuprl, what their level of ability was (novice through expert), and with which of two current versions (Nuprl 4.2 and 5.0) they were more familiar. The free-response part asked what usability strengths and weaknesses Nuprl has on its own and relative to other systems the respondent may have used, and also what features of Nuprl should or should not be changed. The quantitative questions asked the respondent to rate two sets of tasks in terms of how frequently they were necessary and how difficult they were to perform. The possible answers were 1 through 5 (1 meaning "rare" or "easy", and 5 meaning "common" or "hard"). Respondents could also indicate insufficient knowledge by circling "Don't Know"; these answers were discarded.

The tasks were divided into a set of *high-level tasks*, such as "proving a theorem", "editing a term", or "verifying an algorithm", and *basic tasks* such as "deleting a subterm", "understanding editing errors", and "performing proof steps using tactics". We can further subdivide each of these groups into two classes: *core* tasks and *support* tasks that are necessary to perform the core tasks. For example, proving theorems, verifying algorithms, performing basic proof steps, and understanding errors are core tasks, whereas editing terms, managing library objects, and navigating proofs are support tasks. Core tasks are the "real work" that users want to perform using the program, and so support tasks that are necessary but difficult present an obstacle to the usability and the usefulness of the program.

Support tasks are (or should be) conceptually simple and easy to perform, whereas core tasks can be inherently difficult (indeed, they are often difficult problems which the program is intended to help the user solve). Thus, we believe that efforts to improve usability should focus first on difficult and frequent support tasks.

3.2 Respondents

The survey was distributed to 15 researchers and students who have used Nuprl, some of them members of the Nuprl research group, and some non-members. Of these, 9 responses were collected. Five users reported their level of experience as novice or intermediate, and the other four as expert or developer. The respondents also indicated which version (4.2 or the more recent 5.0) they were most familiar with. The experts universally reported version 5.0, while the novices used version 4.2. This correlation is unfortunate, because it means that we cannot distinguish between differences in the survey results that arise because of expert-novice differences and those that arise because of improvements between version 4.2 and 5.0. Six of the respondents reported familiarity with other computational mathematics systems (such as Maple, Mathematica, etc.) and two had experience with other interactive theorem provers (such as HOL, PVS, Twelf).

4 Survey Results and Discussion

For the quantitative responses, we calculated averages and standard deviations for the entire group, the novice group, and the expert group. The results of these calculations are summarized in Tables 1–3. Tables 4 and 5 list the frequency mean, difficultymean , "problem index" (frequency times difficulty), and frequency and difficulty standard deviations for the novices, experts and whole group for the complex and basic tasks. Also listed are the difference rankings (mean novice rating minus expert rating), for frequency and difficulty. We divided the quantitative questions into "positive" and "negative" questions, identified 25 features or concepts that were mentioned either positively or negatively, and counted the number of positive or negative mentions each term received. The results of this analysis are shown in Tables 6 and 7.

For the high-level tasks, novices tended to rate common editing and other support operations as more frequent and more difficult than experts did. This difference seems most likely to be due to learning and mastery of the support tasks on the part of the experts. Conversely, experts rated core tasks as more difficult and more frequent than the novices did. This difference may be because novices get stuck before they have a chance to perform these core tasks, and thus have less experience on which to base their estimates.

The tasks that novice users found most problematic (both difficult and frequent) were term editing, searching (finding previously proved theorems that might be useful), creating theories and definitions, and verifying algorithms. Of these, only "algorithm verification" is a core task. On the other hand, the most problematic tasks for experts were proving various kinds of theorems, writing tactics, and searching for theorems. Of these, only searching is a support task. That searching appears high on the list for both experts and novices suggests that this task may be a serious usability problem.

For the low-level tasks, novices again reported basic editing and support tasks to be more difficult. These tasks were considered easy by the experts, so this is more evidence of a novice-expert difference due to experience. Novices and experts both rated understanding various kinds of error messages as difficult. Some of these error messages are related to support tasks, but some of them are related to performing core tasks such as proofs, and these were reported as among the most difficult to understand. This suggests that the core proving tasks might be easier to perform if better error messages could be written.

The most problematic low-level tasks for novices were creating and editing terms, creating library entries, selecting subterms, and understanding editing errors. All of these are support tasks. For the experts, the most problematic tasks were understanding proof and editing errors, navigating and editing proofs, and using complex tactics. Only "understanding editing errors" and "navigating proofs" are support tasks.

The free-answer questions made it possible for users to comment on Nuprl's inherent strengths and weaknesses, strengths and weaknesses relative to other mathematics programs, and on features which should be changed or kept the same. We categorized concepts and features of Nuprl that were mentioned in a "strength" or "keep the same" question as positive, and those mentioned in "weakness" or "change" question as negative. There were 25 concepts overall, with 22 concepts mentioned negatively, 10 mentioned positively, and seven mentioned both positively and negatively.

The terms that were mentioned most negatively were the nonstandard GUI, the high learning curve, the absence of a Help system, and interface ugliness. The most positive terms included the display form, structured editor, and other technical advantages of Nuprl, as well as flexibility and the ability to display information in multiple windows. The structured editor, display forms, and multi-windowed interface were also mentioned by some respondents as negative, making these areas for possible interface improvement.

5 New Interface Prototypes

Based on the survey results, we have built some preliminary design sketches for a redesigned interface. The three areas that we believe should be improved first are the GUI design, term editing, and the help system. Improving the error messages is also important, but seems to require more careful study of the existing system than the other areas. Figures 4–6 depict some ideas for the overall GUI design and term editor. We suggest replacing almost all routine library commands with menu items and dialogs. We also suggest replacing the heavily modal and keyboard-based term editor with a graphical formula editor like the equation editors in word processors such as Microsoft Word, perhaps with keyboard shortcuts. An initial help system design would simply be to put documentation for all Nuprl expressions, tactics, and rules in a searchable online help system or Web site. In addition, we suggest developing search techniques that allow search criteria other than partial name match.

Currently, many routine operations in the Nuprl interface are accomplished through entering command lines in the ML programming language into the ML top-level loop. For example, to create a new theorem object, one types:

```
create_thm "foo_thm" "foo_obj"
```

where `foo_thm` is the name of the new object and `foo_obj` is the name of another object, before which `foo_thm` will be inserted. Subsequent development of the theorem, including constructing the goal and proof, is accomplished using additional commands such as `view_object`. This interface suffers from the problems of all command-line interfaces, namely that one needs to know everything about a command before one can use it. On the other hand, it shares the advantages of command line interfaces also, in that sequences of commands can be scripted and programmed.

Nevertheless, for many users' purposes, it would be more helpful to supplement or replace the command-line system with a system of menus listing all the available commands, together with dialog boxes or "wizards" for each command requiring additional information. For example, we could replace the theorem creation command with a menu item `New theorem...` which brought up a dialog box that allowed the user to enter the name and goal of the theorem, and to select its position in the list of library objects. The mechanism for performing proof steps uses essentially the same interface. We believe that applying basic GUI design principles to redesign the existing command-line interface could improve usability considerably, especially for novices, for which learning about and memorizing the commands and argument sequences is one of the biggest hurdles in learning Nuprl. One disadvantage of this approach is that most GUIs do not make a provision for user-defined commands and scripting. However, this is not inherent to GUI design; it is possible to build a menu system on top of a scriptable, programmable command system.

Another interface component which presents problems for the novice user is the structured editor ("term editor"). For example, to create a simple term like:

$$x + 1 = \sqrt{y}$$

the user must first "create" a term using Control-O, then enter "eq" for the top-level equality sign, then enter "plus", "variable", "x", "1" for the left hand side, and "sqrt", "variable", "y" for the right hand side. As in the command line interface, one obstacle for new users is the large number of operator construction commands ("eq", "plus", etc) that must be known before one can do simple things. In addition, it can be difficult (at least it has been for me) to recover from certain mistakes in term editing in any way better than deleting the subterm and starting over. For example, once the structured editor has decided that a subterm is a "variable", it is difficult to change this without starting over. The lack of any "undo" or "redo" facility in the structured editor only makes this worse, and although cutting and pasting features are present, they can also be difficult to use.

We suggest that the term editor be redesigned, drawing on existing interfaces that accomplish similar tasks, such as the equation editors in programs like Microsoft Word, or mathematics typesetting languages like LaTeX. Among the most important features of equation editors like Word's is the provision of "palette" windows that display all the available operator symbols and other term constructors. This feature makes it possible for a user to discover capabilities by browsing the palette, rather than searching blindly through (possibly incomplete) documentation. Of course, the palette approach carries its own frustrations, namely it is usually not user-extensible, and often allows no shortcuts to using the mouse to select symbols. Both of these drawbacks can be remedied, and we believe an extensible palette of operator symbols with shortcuts would be much more usable than the existing system.

A key difference between these word-processing approaches to equation editing and the approach needed in Nuprl is that the word processing approaches focus only on outward appearance, while Nuprl needs to allow the user to edit the underlying tree structure of the term, with appearance manipulation a desirable, secondary goal. Our design for an improved term editor makes use of views of the term: an explicit structural tree view (providing information similar to Nuprl's exploded view, but implemented using the tree view interface components found in most modern GUIs), and a rendered, display form view. There are many similarities between Nuprl's underlying term language and existing standards for tree-structured data such as XML, and the process of rendering a term to a display form is similar to the process of rendering an XML document to a displayable form such as HTML or print. Thus, it may be possible to implement this dual structured/display form term editor using standard XML tools, such as the MathML dialect of XML.

At this time there is little current written documentation of Nuprl, and none of it is online. So, for a user to obtain information about a command, tactic, or definition, it is necessary to inspect it directly in the program, consult documentation which may no longer be current, or consult one of the developers of the system. This severely limits the usability of the system outside of Cornell, and by users who prefer to figure things out for themselves. In addition, there is little support for searching for theorems and definitions that are already in the system; although "first match by name" searches can be performed, this requires that one know part of the name of the object one seeks. Short of a successful partial name match guess, the user can only blindly search through the library one page at a time. Consequently, we suggest that Nuprl be augmented with a searchable online help system containing reference material on all commands, tactics, definitions, and other system components, as well as a facility for searching the library on a basis other than object name. For example, it is possible to search through theorems to find all theorems that have something to say about the "plus" operator, or all theorems that mention a specific definition, term or expression. It would be a considerable project in itself to construct a specific design for the kind of help system and search facility we have in mind, so we will not go into further detail.

Once these basic usability improvements have been made to the support tasks, it might be possible to envision new support capabilities that enhance the user's ability to perform core tasks such as theorem proving. For example, one could use the advanced search facility to provide a list of applicable proof steps when the user clicks on a goal; the user could then choose one to try. Indeed, one could imagine making this search feature available as a scriptable command, so that tactics of the form "see if we have already proved this" could be written. However, this is all just speculation at this point.

6 Related Work

There are several papers on usability in HOL[6], a theorem prover that is similar in many ways to Nuprl. HOL is also a tactic-based prover for a form of type theory and higher-order logic that is close to the Nuprl theory. HOL's basic interface is through a command line, but several graphical front-ends to this underlying program have been developed. In contrast, such an approach is almost impossible with current versions of Nuprl, since the command line interface is now hidden underneath a GUI layer. It would be much harder to write additional interface layers on top of this interface. However, Nuprl 5.0 was designed with a separation between the editor (the chief user-interface component) and the rest of the system in mind. It is not clear how feasible it is to develop new editor modules, but it is at least possible that different editors could be implemented.

Thery, Bertot, and Kahn [11] called attention to the usability problem for theorem provers, and showed how to design new interfaces for the HOL system, but did not conduct any formal usability study comparing the old and new interfaces. Aitken et al. [1] did perform a formal usability study in which users of varying expertise were asked to perform a moderately complicated proof using HOL. Aitken and Gray[3] used the GOMS model to analyze definition-making in theorem proving.

Aitken [2] studied the errors made by theorem prover users and classified them into several categories, including knowledge errors (not knowing something useful), memory errors (forgetting the context), and syntactic errors (entering ill-formed data). This study found that knowledge errors and object language syntax errors occurred most among the users in the study. The study identified expression entry, error messages, theory search, and online help/documentation as key areas for usability improvement.

Lowe[8] and Lowe and Duncan[9] describe different versions of a graphical front-end for the CLAM prover called XBarnacle. This work seems to be more conscious of user-centered design. Ozols, Cant and Eastaugffe[10] describes XIsabelle, which is a Unix/X Windows interface to the command-line driven Isabelle theorem prover system. Isabelle, like HOL and Nuprl, is a tactic-based system, so some of the design ideas in XIsabelle might be of use in these systems also. Kadoda[7] presents the results of a case study on the differing views held by designers and users of three theorem proving tools: CADiZ, HOL, and ProofPower. This study also found differences between novices and experienced users/developers; the results suggest that novices tend to see only what the interface presents to them, whereas developers view the interface as a representation of more complex underlying behavior.

7 Conclusions

We performed a study aimed at measuring users subjective impressions concerning the usability of the Nuprl theorem proving system. Based on these results, we have proposed three central areas for improving usability: standardizing the GUI design, simplifying term editing, and providing online help and theory search capabilities. Another key area, error messages, is left open since it is itself a complex issue. Although our results are based on subjective survey responses, they are consistent with objective usability studies of the HOL theorem prover, which is similar to Nuprl in many ways. While not a conclusive verification, this reassures us that the survey results are meaningful.

Many of the elements in our proposed redesign can be built up out of existing components, such as standard GUI menus and dialogs for the basic interface, HTML web pages for the online help system, and structured XML editors and renderers for the term editor/display form viewer. We believe that it would be worthwhile to investigate building theorem provers and other symbolic mathematics systems up out of standard components as much as possible, in order to improve interoperability and reduce the development burden on research groups to reimplement support tasks.

References

[1] J. S. Aitken, P. Gray, T. Melham, and M. Thomas. Interactive proof discovery:an empirical study of hol users. In *Proceedings of User Interface Design for Theorem Provers*, July 1995.

[2] Stuart Aitken. An analysis of errors in interactive proof attempts. Technical report, Glasgow University, 1996. `http://www.dcs.gla.ac.uk/~stuart/ITP/error-paper.ps`.

[3] Stuart Aitken and Phil Gray. On using goms to analyse definition-making in interactive theorem proving. http://www.dcs.gla.ac.uk/~stuart/ITP/goms-paper.ps, 1997.

[4] Allen, Constable, Eaton, Kreitz, and Lorigo. The nuprl open logical environment. In *CADE: International Conference on Automated Deduction*, 2000.

[5] Robert L. Constable et al. *Implementing Mathematics in the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[6] M. J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[7] G. Kadoda. A cognitive dimensions view of the differences between designers and users of theorem proving assistants. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group*, April 2000.

[8] H. Lowe. The use of theorem provers in the teaching and practice of formal methods. In *Proceedings of the Irish Workshop on Formal Methods*, 1997.

[9] H. Lowe and D. Duncan. XBarnacle: Making theorem provers more accessible. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 404–407, Berlin, July 13–17 1997. Springer.

[10] M. A. Ozols, A. Cant, and K. A. Eastaughffe. XIsabelle: A system description. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 400–403, Berlin, July 13–17 1997. Springer.

[11] L. Thery, Y. Bertot, and G. Kahn. Real Theorem Provers Deserve Real User-Interfaces. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 120–129, December 1992. Published as SIGSOFT Software Engineering Notes, Volume 17, Number 5.