Electronic Communications of the EASST
Volume 23 (2009)

Proceedings of the Ninth International Workshop on Automated Verification of Critical Systems (AVOCS 2009)

User Interfaces for Theorem Provers: Necessary Nuisance or Unexplored Potential?

Christoph Lüth

8 pages

Guest Editor: Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

User Interfaces for Theorem Provers: Necessary Nuisance or Unexplored Potential?

Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz, Bremen

Abstract: This note considers the design of user interfaces for interactive theorem provers. The basic rules of interface design are reviewed, and their applicability to theorem provers is discussed, leading to considerations about the particular challenges of interface design for theorem provers. A short overview and classification of existing interfaces is given, followed by suggestions of possible future work in the area.

Keywords: user interfaces, theorem provers, interactive theorem proving

1 Introduction

Theorem provers need to be interactive, and interactive theorem provers need user interfaces. The first part of this statement may sound controversial, but even fully automatic theorem provers need a way to state the proposition to be proven, and the fact of the matter is that any non-trivial proof, be it about program verification, formalised mathematics, or any other application domain, will have to have be conducted with human interaction. So user interfaces are a necessary nuisance, but do they offer more potential?

For most theorem provers, user interfaces have been something of an afterthought in the beginning — understandably so, as developing the core technology was enough of a challenge. With the advances of this technology over the recent years, theorem proving has come of age. The use of theorem proving has spread beyond its previous confines, from case studies to real applications (e.g. in mathematics, software or hardware verification), and with new users the need for better interfaces arises.

In this note, we consider *interactive* theorem provers (the best known examples of which are, in no particular order, Coq, HOL, HOL light, Isabelle, and PVS), which read *proof scripts* containing definitions, declarations, theorems and prover-specific commands. These proof scripts are the central artefacts under construction; supporting the user to interactively construct them should be the main purpose of the user interface. Even though phrased in terms of theorem proving, the discussion also pertains to interactive formal method tools; we come back to this in the conclusions.

We first consider criteria for a good user interface for theorem provers, and then review existing interfaces and their strengths and weaknesses. From this, we point out some directions of future research, and conclude with the major challenges and a résumé.

(i) Strive for consistency.
(ii) Cater to universal usability.
(iii) Offer informative feedback.
(iv) Design dialogs to yield closure.
(v) Prevent errors.
(vi) Permit easy reversal of actions.
(vii) Support internal locus of control.
(viii) Reduce short-term memory load.

Figure 1: The 'Eight Golden Rules' of interface design. Taken from [SP09, p. 88f].

2 What makes a Good User Interface?

2.1 General Criteria

Figure 1 shows the 'golden rules' of interface design according to Shneiderman [SP09] (other authors give similar guidelines). They are all relevant for interactive theorem proving, but some rules particularly so: the sixth, because interactive theorem proving by its very nature is an explorative process, so it is important to be able to try and undo proof steps; the eighth, alluding to the seven (plus or minus two) chunks of information that can be held in short-term memory, because theorem provers can actually offer a lot of information to the user, and the problem is to avoid overwhelming the user; and the second, because users will range from experts who know exactly what they want and how to achieve it and might prefer a programmable command line interface, to complete novices who need every help they can get, and prefer syntax-free interface elements such as menus.

On the other hand, the seventh rule should be taken with a grain of salt. It means that it should be easy to get the system to do what the user wants to achieve, which in a theorem prover means proving propositions. Unfortunately, keeping the user from proving wrong propositions is the core of interactive theorem proving, and the resulting frustrations are par for the course (and cannot be blamed on the interface). Showing *why* a particular action does not work, on the other hand, is an important task of the interface, so the rule should be read in this sense.

2.2 Challenges in User Interfaces for Theorem Provers

Theorem provers are special programs, and designing a good interface for them offers special challenges. The first difficulty is that theorems or proofs are abstract in the sense that they have no physical counterpart. Hence, *syntax* plays a central rôle in theorem proving (and mathematics), because this is what is being manipulated. The capability to read and write proofs in a notation which is close to what users are used to from text books cannot be overstated, because it eases the cognitive load on the user considerably.

Secondly, theorem proving is very hard, and proof scripts are very abstract in the sense that they condense much information; they cannot be manipulated as conveniently as e.g. source code. In particular, the high degree of interdependency tends to make proof scripts brittle, so changing them in one place may lead to unexpected failure elsewhere, which makes changing

and maintaining larger proofs very frustrating.

Thirdly, theorem provers potentially offer a lot of information: the proof state can become very large, the amount of rules, theorems, proof procedures known to the system can run into thousands, etc. It is important not to overwhelm the user, but it is even as important (and more challenging to implement) to allow the user to query the system interactively, preferably at different levels of abstraction, depending on the user's proficiency.

The most important consequence of these considerations is that user interface and theorem prover need to interact closely, with control flow going in both directions; interface design for theorem prover is more than 'bolting a bit of Tcl/Tk onto a text-command-driven existing prover in an afternoon's work' [BS98].

3 A Review of User Interfaces Past and Present

3.1 The Early Days

In the early days, interactive theorem provers were used from the command line. Users wrote a proof script which they fed to the prover, and the prover would check it; the interaction was in batch mode, very much like a with a compiler. Although today this modus operandi would be considered unproductive, it was standard practice back then. Moreover, the interactive theorem proving community has never been large, and subsequently resources to develop user interfaces have always been scarce (comprised to no small part of postgraduate students struggling to produce a thesis under the limitations of the prevailing user interfaces, and gratefully finding some justifiable diversion from their thesis work).

3.2 Emacs and Proof General

Under these limitations, the Emacs editor, which allows for comfortable and flexible customisation using the Lisp dialect it is written in, offered an excellent platform, and soon specialised Emacs modes for many of the popular provers appeared. After a while, it became clear that many provers shared a similar interaction mode, and maintaining each of them separately was an unnecessary burden. The Proof General project [Asp00] in Edinburgh consolidates the different Emacs interfaces for Isabelle, Lego and others into one Emacs package which can be instantiated to the different provers.

Proof General found widespread use and is the most popular interface implementing the idea of *script management* introduced in [BT98], where the proof script is treated as a sequence of commands, which are processed in a linear fashion. This divides the script in three regions, one of which has already been processed, one of which is currently being processed, and one of which is unprocessed. Once a region has been processed, it can not be edited anymore. A simple undo function allows the user to go back in the proof. This idea is strikingly simple and powerful; it is cheap to implement on the prover's side, and on the other hand offers a flexible way to add more functionality in the user interface (e.g. a 'go to here' button, which performs forward or backward steps as required).

3.3 Integrated Development Environments

An integrated development environment (IDE) offers a tight integration of source code editor, compiler, debugger, documentation browser, and other tools. SmallTalk was the first language to come with an IDE, and they became really popular with Borland's Turbo-Pascal. With the similarities between theorem proving and software development (in both, the object of interest — source code and proof script respectively — is processed by an external tool — compiler or theorem prover), it seems tempting to construct an IDE for theorem proving [TBK92]. Early attempts include CtCoq and PCoq [ABPR01]. These attempts have been hampered by the fact that the integration between a theorem prover and its interface needs to be far closer than between a compiler and a source-code editor, and that in particular maintaining an IDE is a substantial task — many of these efforts fall out of use because the underlying prover changes and is no longer compatible with the interface, or because they were built using an interface toolkit which has fallen out of use. Recently, more powerful toolkits made it easier to create IDEs, such as the CoqIDE created using GTK+, but this is still the exception rather than the rule.

3.4 Graphical User Interfaces

Graphical user interfaces (GUIs) entered the scene as early as the 70s with the Xerox Star system. The methodology behind graphical user interfaces is *direct interaction*: all objects of interest are represented continuously and graphically, preferably using an understandable metaphor, and can be manipulated with syntax-free operations on this representation, such as pointing at them, moving them, or causing interaction by dropping them onto other objects.

Finding such a metaphor for theorem provers is a challenge, since the objects in a theorem prover are abstract, and it is far from clear how their manipulation can be modelled by intuitive gestures, although attempts have been made [LW99]. The Jape system was a pioneering effort [BS99]; it was designed to be a 'quiet interface', meaning it would only show as little as needed and not as much as possible (which as pointed out above is good interface design practice), and uses gestures to select proof steps.

3.5 Document-Centered Approaches

The PVS system has developed a closer interaction model with the Emacs editor then the other systems mentioned in Sect. 3.2. The user is essentially editing an interactive document in the editor's buffer, with the prover checking the semantic integrity in the background. This is the so-called *document-centered* approach, where the focus of attention is the proof script itself, and how to edit it, rather than it being processed by a prover. It works best with a style of proof scripts which is not a simple sequence of state-affecting prover commands, but where the proof script represents the proof itself, e.g. by stating a sequence of transformations or intermediate goals. The Mizar prover pioneered this approach [T+73], and Isar brought it to the Isabelle system [Wen99]. Taking this idea one step further is the Plato system [WAB06], which uses the Texmacs editor to provide WYSIWYG editing of mathematical documents in a LaTeX-like language with high quality typesetting, while the proofs are checked by the Omega-prover in the background.

4 The Future of User Interfaces

What have the interfaces introduced in the last chapter achieved? Without wishing to denigrate the efforts of the researchers involved, there is still a lot of room for improvement. What we can take from the existing interfaces is that as Proof General shows, it is good to be generic. Hardly any theorem prover has a large enough developer base to develop its own interface, but by sharing the effort across different provers we can achieve something. Genericity is also good because it helps to make the connection between interface and prover clear; e.g. the interaction protocol for Proof General was made explicit in the PGIP protocol [ALW07]. It is also important to note that the success of a prover hinges mainly on its expressiveness and proof support; in the past, users have always preferred a powerful prover with an Emacs interface over a less powerful prover with a slick GUI, even if the latter is easier to use. The aim must be, then, to provide existing powerful provers with better interfaces.

4.1 Modern IDEs

Early attempts to develop IDEs for theorem proving have been mentioned above. With modern IDEs such as Eclipse and NetBeans which are specifically designed to be generic, the situation has improved, and it is tempting to instantiate e.g. Eclipse as a theorem proving interface [ALWF06]. However, Eclipse is not exactly light-weight, and a major disadvantage of most IDEs is that they do not support mathematical notation well.

Particularly appealing in Eclipse is its incremental document processing. That is, there is no explicit 'process this document' step, rather the prover (or compiler) continuously processes as much of the document as possible in the background, flagging up errors as they occur. This asynchronous mode of interaction makes good use of the time the user spends thinking, increasing overall responsiveness of the system.

4.2 Emerging Technologies

The most drastic change in interface technology over the last years has possibly been the rise of web-based technologies. The technique known as AJAX (asynchronous Java script and XML) has taken web-based interfaces from filling in forms to fully interactive graphical user interfaces, and in future the distinction between local (desktop-based) and remote (web-based) interfaces will probably be blurred even further. These technologies can play a rôle in theorem proving too, as they allow easy cross-platform access to a theorem prover without having to install it locally, often a daunting task for the novice. An impressive first step here is Kaliszyk's ProofWeb [Kal07].

4.3 Interaction Models

There have been various attempts to adapt more intuitive interaction models like gestures into theorem proving interfaces, like in Jape or Coq ('proof-by-pointing' [BKS97]). It seems tempting to allow the user to rearrange formulae by drag-and-drop, going beyond what pen-and-paper mathematics allows us to do. However, this has to be reconciliated with the fact that the main artefact of a theorem prover is the proof script; a proof consisting of a series of gestures is not

really useful. Thus, gestures should be seen as a way to create proof scripts. Users indicate that they wish to perform induction on x, or exchange the two arguments of $+$ (and this can be done either via drag-and-drop gestures, a menu button, or even more exotic means), the prover returns a new proof script fragment, which the interface inserts into the proof script. The challenge is to provide a uniform interaction protocol which works reliably across different provers (a first attempt has been made in [ALW06]).

4.4 Foundations

Interfaces have mostly been seen in technological terms. This is understandable, because technology delivers to the user, but the theoretical foundations of user interfaces have not received much attention. An exception is Denney's work [DPT05], which introduced the notion of *hierarchical proofs*, and operations such as zooming into a proof, on a purely semantics-free level. This allows interfaces to implement operations on this level, separating the purely syntactic manipulation which can be done in the interface from the semantic manipulations in the theorem prover.

5 Conclusions

We have highlighted the challenges in constructing interfaces for theorem provers, reviewed existing interfaces, and pointed out some directions of future research. All of this is necessarily subjective, so the author is grateful for any omissions pointed out to him. The discussion here has been phrased in terms of interactive theorem provers, but applies equally well to formal methods tools; the key difference between formal method tools and theorem provers is that because formal method tools typically have a more singular purpose (e.g. proofs in a particular notation or of particular properties), users and their level of proficiency will be less diverse, but the points about consistent notation and genericity are equally valid.

As a closing summary, the key technical challenges in the author's estimate are the comprehensive support of mathematical notation (maybe standard vector graphics formats such as SVG can offer a solution here), and a clear standard protocol for theorem provers to interact with user interfaces. PGIP was a first start in this direction, but possibly it is oriented too much towards script management; a recent new version [AALW09] aims to rectify this shortcoming.

The overall challenge in user interfaces is to leverage the underlying technology to an extent which makes it *easier* to do proofs in a computer than with pen and paper. Presently, this is not the case. Theorem provers tend to get in the way more often than they are helpful, and even though that is in part their duty as proof checkers, the preferable rôle model of a theorem prover should be that of a helpful co-author gently pointing out errors and suggesting improvements, rather than a stubborn civil servant refusing to accept the blindingly obvious because of some formality. In good part, this an interface issue, and hence the author's answer to the initial question is that there is definitely unexplored potential, waiting to be developed by enterprising minds.

Acknowledgements: Research in part supported by the German Research Agency (DFG) under grant LU 707-2/2.

Bibliography

[AALW09] D. Aspinall, S. Autexier, C. Lüth, M. Wagner. Towards Merging PlatΩ and PGIP. In *Proc. 8th International Workshop on User Interfaces for Theorem Provers (UITP 2008)*. Electronic Notes in Theoretical Computer Science 226, pp. 3– 21. Elsevier Science, 2009.

[ABPR01] A. Amerkad, Y. Bertot, L. Pottier, L. Rideau. Mathematics and Proof Presentation in PCOQ. In *Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01), Sienna, Italy*. 2001. also available as INRIA RR-4313.

[ALW06] D. Aspinall, C. Lüth, B. Wolff. Assisted Proof Document Authoring. In Kohlhase (ed.), *Mathematical Knowledge Management MKM 2005*. Lecture Notes in Artificial Intelligence 3863, pp. 65– 80. Springer, 2006.

[ALW07] D. Aspinall, C. Lüth, D. Winterstein. A Framework for Interactive Proof. In *Mathematical Knowledge Management MKM 2007*. LNAI 4573, pp. 161– 175. Springer, 2007.

[ALWF06] D. Aspinall, C. Lüth, D. Winterstein, A. Fayyaz. Proof General in Eclipse. In *Eclipse Technology eXchange ETX'06*. ACM Press, 2006.

[Asp00] D. Aspinall. Proof General: A Generic Tool for Proof Development. In Graf and Schwartzbach (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science 1785, pp. 38–42. Springer, 2000.

[BKS97] Y. Bertot, T. Kleymann, D. Sequeira. Implementing Proof by Pointing without a Structure Editor. Technical report ECS-LFCS-97-368, University of Edinburgh, 1997. Also published as Rapport de recherche de l'INRIA Sophia Antipolis RR-3286.

[BS98] R. Bornat, B. Sufrin. Using gestures to disambiguate unification. In *User Interfaces for Theorem Provers UITP'98*. 1998.

[BS99] R. Bornat, B. Sufrin. A minimal graphical user interface for the Jape proof calculator. *Formal Aspects of Computing* 11(3):244– 271, 1999.

[BT98] Y. Bertot, L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation* 25(7):161–194, Feb. 1998.

[DPT05] E. Denney, J. Power, K. Tourlas. Hiproofs: A hierarchical notion of proof tree. In *Proceedings of Mathematical Foundations of Programing Semantics (MFPS)*. Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, 2005.

[Kal07] C. Kaliszyk. Web Interfaces for Proof Assistants. In Autexier and Benzmüller (eds.), *Proc. User Interfaces for Theorem Provers (UITP'06)*. ENTCS 174(2), pp. 49–61. 2007.

[LW99] C. Lüth, B. Wolff. Functional Design and Implementation of Graphical User Interfaces for Theorem Provers. *Journal of Functional Programming* 9(2):167– 189, Mar. 1999.

[SP09] B. Shneiderman, C. Plaisant. *Designing the User Interface*. Addison-Wesley, 5th edition, 2009.

[T+73] A. Trybulec et al. The Mizar Project. 1973. See web page hosted at http://mizar.org, University of Bialystok, Poland.

[TBK92] L. Théry, Y. Bertot, G. Kahn. Real theorem provers deserve real user-interfaces. *SIGSOFT Softw. Eng. Notes* 17(5):120–129, 1992.

[WAB06] M. Wagner, S. Autexier, C. Benzmüller. PLATΩ: A Mediator between Text-Editors and Proof Assistance Systems. In Autexier and Benzmüller (eds.), *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*. ENTCS. Elsevier, 2006.

[Wen99] M. Wenzel. Isar — a Generic Interpretative Approach to Readable Formal Proof Documents. In Bertot et al. (eds.), *Theorem Proving in Higher Order Logics TPHOLs'99*. Lecture Notes in Computer Science 1690, pp. 167– 184. Springer, 1999.