Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 103 (2004) 139–159

www.elsevier.com/locate/entcs

Thoughts on Requirements and Design Issues of User Interfaces for Proof Assistants

Norbert Völker[1]

Department of Computer Science
University of Essex
Colchester, United Kingdom

Abstract

This position paper discusses various issues concerning requirements and design of proof assistant user interfaces (UIs). After a review of some of the difficulties faced by UI projects in academia, it presents a high-level description of proof assistant interaction. This is followed by an exposition of use cases and object identification. Several examples demonstrate the usefulness of these requirement elicitation techniques in the theorem proving domain.
The second half of the paper begins with a consideration of the "principle of least effort" for the design of theorem prover user interfaces. This is followed by a brief review of the "GUI versus text mode" debate, proposals for better use of GUI facilities and a plea for better support of customisation. The paper ends with a discussion of architecture and system design issues. In particular, it argues for a platform architecture with an extensible set of components and the use of XML protocols for communication between UIs and proof assistant backends.

Keywords: User interfaces, theorem provers, requirements engineering, HCI design, reusability, component architectures.

1 Introduction

Theorem proving environments such as COQ [27], HOL [13], Isabelle [23], Nuprl [1], Omega [26], PVS [24] and others [28] are becoming increasingly popular for the formalisation of derivations in different areas of computer science and mathematics. Some of these systems are associated with user interfaces (UIs) that try to alleviate the strain of dealing with a complex proof assistant. Recent noteworthy achievements include ProofGeneral [9],

[1] WWW: `http://cswww.essex.ac.uk/staff/voelker`

doi:10.1016/j.entcs.2004.05.001

Pcoq [2], $\mathcal{L}\Omega\mathcal{UI}$ [25], the Nuprl interface and IsaWin [19]. Nonetheless it is fair to say that progress in user interfaces has been relatively slow compared to advancements in the proof assistants themselves.

The aim of this paper is to aid the development of future UIs by discussing important issues concerning requirements elicitation, human computer interface (HCI) design, architecture and system design. Emphasis will be put on reusability and adaptability as these are seen as key design goals.

2 Difficulties of Prover UI Projects in Academia

Most leading proof assistants have been around in some form or other for a decade or longer. Several have matured to a point where they are suitable for mechanical proof checking of substantial derivations.

The gradual development of proof assistants contrasts with the more haphazard history of theorem prover user interfaces. A look at the literature of the 1990s reveals several projects which seem to have had little impact. In some cases, the projects produced systems that never advanced beyond rather small groups of users. It is interesting to see that among the more popular survivors are several systems based on (X)Emacs, i.e. ProofGeneral and the user interfaces for PVS and IMPS [29].

In order to increase the impact of future proof assistant UI efforts, it is wise to be aware of some of the difficulties facing such projects.

- Constructing a good user interface for any complex system is an elaborate task that requires good skills both in HCI and program design. In the case of theorem prover UIs, there are still open questions concerning the appropriate organisation of the interface and how the various items are best visualised.
- Implementing a user interface can be a time-consuming task that might be perceived as less interesting compared to developing the proof assistant itself. It is easy to underestimate the necessary efforts because of the apparent visual simplicity of the end product.
- Systems that are built on top of a platform such as (X)Emacs require less coding. However, they can be plagued by platform restrictions and problems which are out of the developer's control. Also the addition of features to the platform sometimes requires coding in custom scripting languages. This makes it less likely to get support from other developers.
- There is little external reward for building and maintaining such user interfaces. Because of the lack of commercial interest, most developers are academics or students. Members of both of these groups are usually not paid

for perfecting and maintaining user interfaces. Instead, their reward for this type of work is only indirect, namely in so far as it contributes to research success, teaching, and, in case of students, the award of qualifications.

- Most theorem provers have been developed in functional programming languages from the ML and Lisp families. Compared to other languages such as C++, Java or Basic, these functional languages are lacking in graphical user interface (GUI) toolkits and in some cases provide only poor support for light-weight concurrent processes. These problems have lead to several projects that connect functional languages with frameworks that have better GUI facilities, see for example sml_tk [20] and recent OCaml [30] bindings for TK and GTK+. Still, the resulting development environments are more cumbersome than the direct use of a modern commercial GUI builder tool.
- Theorem provers are still rapidly evolving and so are user interface technologies. This increases the required maintenance effort.

For the reasons mentioned above, only few academic institutions have the necessary resources to build and maintain completely custom UIs. Even these institutions would benefit from more reuse by freeing up developer time. This suggests that it is in the interest of the academic theorem proving community to devise reusable UI components and UIs.

3 Proof Assistant Interaction

In the past, with the notable exception of ProofGeneral, most UIs have been targeted at a particular proof assistant. Unless great care is taken, this leads to a design with a tight coupling between the UI and the proof assistant. This makes reuse difficult. Although doubtlessly influenced by the author's experience with Higher-Order Logic theorem proving environments, this paper tries to approach UI requirements from a more general perspective.

The primary role of the UI is to provide support for the development of theories with a proof assistant. Here is an outline of the interaction from a user point-of-view:

- Development usually takes place within some *project* such as building a library, finding a proof for a mathematical theorem or the formal verification of a program.
- Development is partitioned in a number of *theories* (or "sections"). Theories are hierarchical: each child theory extends one or more parent theories.
- Theory developments can be made persistent, for example in form of proof script files. The system provides commands to store and load theories.

- Theory development activities take place in a *logical context* consisting of
  - · a *logical basis* that comprises declared constants, axioms, and for typed logics the declared types and possibly sorts;
  - · the set of *theorems* that have already been proven for this logical basis.
- Apart from the logical context, there is also an *extra-logical context* that influences the user's interaction with a system. This includes the setup and current state of parsers, pretty printers, proof tools, display options and documentation generators.
- An interactive *proof* is started by posing a logical formula (the *main/ initial goal*) that is to be proven. This is followed by a number of *proof steps.* In each step, the user analyses the proof state and then issues some *proof command.* This is carried out by the prover and leads to the next *proof state.*
- A proof state is characterised by a number of *open goals* that are left to prove. The proof state might also have other features such as *local declarations* and *assumptions.*
- The proof of the initial goal is finished when a state is reached that has no open goals.
- Proof attempts can be aborted.
- On some systems, it is possible to temporarily abandon a proof attempt and return later to this point in the proof. This makes it possible to *nest proofs* whereby a user interrupts a proof attempt and first derives some auxiliary theorems that are needed for the continuation of the current proof.
- Proof commands can refer to proof tools, theorems, goals, types, terms and other entities.

The term "theory development" does not just denote the expansion of theories with new constants, theorems and other elements. It also refers to activities that concern the modification, inspection and deletion of theory elements. For example, this could be the inspection of the current proof state, the editing of an existing proof or the deletion of a theorem.

4 The Case for Requirements Elicitation

In the author's experience, academic projects do not often carry out a detailed requirements elicitation. Writing requirements documentation might be perceived as a job which ultimately takes more time than it is worth. This section argues for the usefulness of use cases and object identification as two relatively lightweight means of requirements elicitation.

First Use Case Example: Adding a constant to a theory

Use case analysis [16] is one of the more successful software engineering technique for collecting requirements. Each use case abstracts over a set of user-system interactions ("scenarios") that are performed by users in pursuit of a certain aim.

Below is a sample use case for a hypothetical proof assistant system. It deals with the addition of a new logical constant to an existing theory. This is a task that typically arises during the iterative development of a theory when one notices at some point that it would be useful to introduce a further constant. The formulation is based on a use-case template by Larman [18].

Title of Use Case: Adding a constant to a theory

Actors: Specifier

Preconditions: System is either in top-level or theory mode.

Postconditions (Successful Outcome): Constant has been added to the theory

Main Success Scenario:

1. The specifier requests "Add constant" operation.
2. The system responds with a list of theories for the current project.
3. The specifier selects a theory.
4. The system responds with a display of the constants already declared in the theory and a form for entering the details of the new constant.
5. The specifier enters the constant details (usually type and definition) and submits the form.
6. The system adds the constant to the theory.
7. The system indicates the successful completion of the operation by a status message in the UI.

Alternative Flows:

1-3 a. If the system is in theory mode, then the specifier can also request "Add constant to current theory". In this case Step 2 and Step 3 are omitted.

2b/4 a. The specifier can cancel the process by choosing a "cancel" option.

4 b. If the specifier indicates the theory by string input, then the system checks that the current project contains such a theory. If this check fails, then the system responds with a suitable error message and redisplays the list of theories.

4 c. The system checks that the specifier is authorised to change the logical basis of the theory. If this check fails, then the system responds with a suitable error message and redisplays the list of theories.

6 a. The system checks that the input from Step 5 specifies a valid new constant for the theory. If this check fails, then the system responds with a suitable error message and redisplays the constant details input form.
7 a Cancellations and errors are signalled by appropriate status messages.

Special Requirements:
· All text input forms should allow copy-and-paste.

Open Issues
· Should there be a syntax-directed editor for input of the constant details?

Actors

In a use case, the term "actor" refers to a particular role in which users interact with a system. The actor "specifier" above originates from an article by J. Goguen [12] who distinguishes between the three actors "specifier", "prover" and "reader". These three can all be seen as sub-roles of a more general "theory developer" role. Other roles in which users might interact with a proof assistant are a "proof tool developer" actor who performs activities such as programming, adaptation and testing of proof procedures as well as the standard "system administrator" actor who performs activities such as installation, monitoring and updating of the system. Lastly, it can be useful to introduce sub-roles that distinguish users with different experience, i.e. there might be variations in use cases for "novices" compared to "experts".

Second Use Case Example: Performing a Proof Step

The use case example above was relatively low level. In particular it did not refer to other use cases but was complete in itself. For contrast, here is a more complex hypothetical use case which potentially involves carrying out other "sub" use cases. It is assumed that like in PVS, the theorem proving environment automatically keeps track of proof steps. In order to avoid confusion, the actor will be referred to as "user" rather than "prover".

Title of Use Case: Performing a step in an interactive proof

Actors: User in role "performing proof".

Preconditions: System is in proof mode. This implies that the theorem to be proven has been set as the initial goal. An unspecified (possibly zero) number of proof steps have already occurred.

Postconditions (Successful Outcome): The proof is one step closer to completion.

Main Success Scenario:

1. The user analyses the current proof state and decides on the next proof command.
2. The user enters the next proof command. This can be done by selection with a pointer device or via textual input.
3. The system carries out the proof command.
4. The system displays the new proof state and prompts the user for the next proof command.

Alternative Flows

1 a. The user might require further information before deciding about the next proof step. This could mean searching for suitable, already established lemmas, browsing a list of proof commands (possibly suggested by the system), browsing the proof history, browsing the previous steps of the current proof, searching for a similar proof, or consulting online documentation. These activities are themselves new use cases.

1 b. The user might decide to abort a proof, for example because the initial goal is unprovable and needs to be changed. In this case the user enters a proof abortion command in Step 2.

2 a. The user can also enter commands which do not change the proof state but which otherwise affect the state of the proof assistant or the user interface.

3 a. The proof state remains unchanged if the proof command is illegal or fails.

3 b. The user can request cancellation of the proof command which is currently being executed by the system. If cancellation is successful, then the proof state will remain unchanged.

4 a. The system displays appropriate messages in case of illegal or failed proof commands, a successful completion of a proof by proving all subgoals, warnings during proof step execution, proof step cancellation and abortion of proofs. In case of failed commands, the system should also provide feedback.

Special Requirements

· None

Open Issues

· In certain situations, could the system try to finish a proof itself while waiting for input from the user, and if successful alert the user of this fact?
· Should the system support nested proofs?
· In Step 3, should the system provide an indication of progress?

The use case example above shows the limits of the high-level approach. In a real UI development project, developers would make the different use case steps more concrete, for example by detailing how the search for auxiliary theorems should be supported by the system. Similarly, support for the invocation of particular proof methods depends of course on the facilities offered by the theorem prover. The second "open issue" additionally demonstrates the difficulty of cleanly separating UI and theorem prover requirements, as the nesting of proofs could in principle be supported by either.

Despite these limitations, the abstract use case highlights weaknesses of some current systems. Support for the lookup of auxiliary theorems is often not as efficient as one would hope, given that this tends to be a frequent user activity. The author is also not aware of an implementation of the suggestion in the first "open issue" in any of the popular interactive prover/UI combinations.

Identification of Domain Objects

An important step in requirements engineering is the identification of (classes of) domain objects that are relevant from the user's point of view. A simple starting point to finding such objects are the nouns in use cases and other descriptions of user-system interactions. In our case, object classes can be found by going through the description in Section 3 and by looking at proof assistant manuals. Below is an initial, incomplete list of domain objects that are relevant to the interaction of users with a proof assistant:

- project, theory, theory hierarchy, child theory, parent theory;
- logical context, logical basis, extra-logical context;
- axiom, theorem;
- formula, term, constant, type, sort;
- proof, goal, main goal, proof step, proof command, proof state, open goal, proof attempt;
- tools such as parsers, pretty printers, proof editors, natural language processors, and so on.

For good reasons, most theorem proving systems are written in functional programming languages. Hence the reader might wonder if there is a problem here as the term "objects" might be interpreted as a bias towards implementation in an object-oriented programming language. However, while it is well-established that such programming languages are very suitable for coding user interfaces, there is no suggestion here that this task could not be done in other kinds of languages. The main emphasis in the identification of objects is on the classification of domain elements into separate categories that are used as the basis of further development. In case of functional programming languages, these categories will be implemented as types. In this context, it is instructive to note that the code of modern theorem prover systems is organised around structures that group together types and their associated operations. This is analogous to the organisation of object-oriented programming languages around classes.

Finding Use Cases

Domain objects help to structure the search for use cases. For each class of objects, one can consider in how far there is a need to support operations from the following generic categories:

- creation of objects,
- modification of objects,
- inspection of the state of objects,
- making objects persistent,
- deletion of objects.

Application of this technique quickly leads to lists of use cases for theorem proving environments. For example, in case of the domain class "theorem", it gives rise to the following use cases:

- Creation of a theorem via an interactive "backwards" proof.
- Creation of a theorem directly — without stating a goal — by applying operations to existing theorems.
- Renaming a theorem.
- Editing the proof of a theorem.
- Modification of the statement that is proven in a theorem. This might also require an adaptation of the proof.
- Inspection of a theorem and its associated proof.
- Saving a theorem with its associated proof.
- Deletion of a theorem from a theory.

Here are three more use cases which were identified in this way and which lack support in the current ProofGeneral/ Isabelle combination:

- While Isabelle provides a number of operations to inspect and modify the state of proof tools, there is no support for such actions in ProofGeneral.
- Renaming a constant is a use case that occurs occasionally during the initial phases of the development of a new theory. Neither ProofGeneral nor

Isabelle supports this activity directly. Instead, the user has to manually edit all occurrences and rerun proof script files.

- There is no support for use cases involving projects in ProofGeneral such as creating a new project, moving theories from one project to another, or "making" a project (running all proofs and associated document generation). In fact, there is no explicit notion of projects in Isabelle — they are identified with directories.

These three use cases illustrate different problem situations. In the first case, the UI facilities are simply lagging behind the capabilities offered by a particular proof assistant. This could be mended easily by adding custom UI elements that invoke these operations. In the second example, the use case is not supported directly by the proof assistant. Hence support in the UI would also require an extension of the proof assistant. In the third case, a whole concept ("project") is missing from the UI/ proof assistant combination.

The Role of Use Case Analysis

Use cases are a relatively simple and lightweight requirements elicitation technique. Their main purpose is as a means of communication and documentation of functional requirements. Typically, they are written during the early phases of software projects, be it the development of new systems or the extension of systems with new functionality.

Use cases provide a good starting point for user interface design, testing and — in case of theorem provers much needed – user documentation. Use cases can be complemented by other fact-finding techniques such as questionnaires, interviews or observation of users of existing systems.

One can expect the following deliverables from a use case analysis of a theorem prover assistant/UI combination:

- A list of domain classes that play a role in user interaction.
- A catalogue of use cases structured according to domain classes. This catalogue would normally only list the use case titles. The entries should be annotated with priorities, for example by distinguishing primary, secondary and optional use cases.
- A detailed description of key use cases based on a template such as used in the two examples above above. These descriptions can be structured using "generalisation", "includes" and "extends" constructs [4].
- Optionally, a graphical overview of use cases in form of a UML use case diagram.

Use case analysis is no panacea that solves all problems of requirements engineering. For example, the identified domain classes are not necessarily appropriate as design and implementation classes. This point was made by B. Meyer [22] who also criticised use cases for their specification of sequentiality. Another problem is that use cases might invite a rather low level, concrete view of the system that is too much influenced by similar, already existing systems.

Despite these drawbacks, experience with use cases over the last decade has shown that they are a valuable technique for collecting requirements. For the development of theorem proving environments, the author believes that perhaps their most important benefit arises from the fact that use cases are user-centric. This should help as a counterbalance in what otherwise tends to be a prover-functionality driven development. Another strength of use cases is that they identify sequences of interactions which need to be supported efficiently as a unit. In the author's experience of proof assistants, supporting whole use cases rather than isolated interactions is a point that does not always seem to be taken into account.

5 Human Computer Interface (HCI) Issues

HCI Design: Principle of Least Effort

The interaction of a user with a proof assistant is an instance of human-computer interaction (HCI) [10]. As such, the usual guidelines of good HCI design apply. One basic rule is the "principle of least effort". According to this rule, the interface design should be such that users can perform their tasks with as little effort as possible. Here are some steps towards this aim:

- The amount of required typing can be minimised by the use of keyboard shortcuts, menus, tool bars and auto completion. Systems might support input using a shorthand version of commands that can be expanded later into a more readable format. Reduction of input effort is an important issue as theorem prover interaction can require a lot of typing and mouse/pointer operations. Like programming, this can lead in extreme cases to the risk of developing repetitive strain injuries (RSI).
- Whenever possible, the system should offer choices from which a selection can be made. This has well-known advantages such as making the user aware of different options and making it unnecessary to memorise the concrete syntax of commands and the precise names of entities. In case of theorem prover UIs, this can be achieved for example by providing tool bars with buttons for frequent proof commands or allowing the identification of theorem parameters in proof tactics by selection rather than typing

names.

- Auto completion could be used whenever it is necessary to name objects such as theories, theorems, proof commands, etc. Completion should be intelligent in the sense that it takes object types and context into account. An advanced implementation could perhaps even consider the likelihood of different possible completions.
- Hiding of irrelevant information decreases the amount of effort in finding information and making decisions. For example, menus should offer only possible choices. When searching for theorems in a library, it should be possible to do so without being distracted by the proofs of these theorems.
- Sorting items according to the likelihood of their usefulness is another way of decreasing effort. For example, the user interface could keep track of how often certain menu items are used. Rarely-used menu-items can then be hidden and only made visible on explicit demand.
- Use of suitable default values whenever an input choice is offered.
- Allow reuse of previous inputs, i.e. base a new constant definition on editing a similar previous definition or allow editing of a previous proof command.

K. Eastaughffe [11] has identified further HCI design principles for theorem prover support. In particular she suggests complementary views of proof constructions, ease of undo operations, flexibility in the way users can articulate commands to a prover, and support for concurrent proof constructions.

Graphical Interaction versus Text Mode

Graphical User Interfaces (GUIs) have transformed the use of computers and contribute significantly to their popularity. They are kind to novices and non-experts as they make it unnecessary to learn command languages. GUIs are based on the direct manipulation metaphor: objects are represented by graphical elements such that a manipulation of the graphical representation induces a corresponding operation on the represented object. A particular strength of GUIs is the manipulation of complex objects. For example, visual editors have made command line text editors obsolete. The latter are typically only used in emergency situations.

Despite all the arguments for GUIs, the case for graphics-based interaction is not always clear cut. Text-based interfaces can be more efficient — selecting an item from a long scroll list or deeply nested menus can take longer than simply typing its name, especially when auto-completion is provided. In the case of theorem prover UIs, there are conflicting reports. Some users seem to like proof trees [15] such as provided by PVS or Jape while others have commented that — while useful for pedagogical purposes — such trees can be difficult to use for all but the smallest proofs [12]. One study has even cast general doubt on the usefulness of GUIs for proof assistants [21]. In particular, it mentions the danger of brittle proofs arising from the identification of subterms by pointing and comments that "There is even strong evidence to suggest that ease of interaction tends to reduce planning and lead to poorer user performance in problem solving domains...".

In our view, this controversy suggests that the UI should offer a choice between text-based and graphical interaction whenever there is no clear-cut advantage to use either. This applies, for example, to selections which are often faster by keyboard, at least for expert users. Providing such a choice also caters for individual preferences.

Better Use of GUI Facilities

Because of the potential benefits, there is a strong case for continuing research into better use of GUI facilities during proof assistant interaction. Especially promising is the use of hyper-linked text as it cuts the time of finding related information. Here are some potential applications:

- Theorem search results: link to proofs.
- Constant in a goal: link to its definition and theorems where it occurs.
- Browsing proofs at a high level with hyperlinks that uncover the details.
- Unknown in a goal: link to a box which allows its instantiation.
- Links between closely related theorems.

Drag-and-drop operations are useful for manipulating "container" objects and aggregations. In the context of proof assistants, possible applications include:

- Reorganising theories by movings subsections from one theory to another.
- Assembling theorem sets that parameterise a proof procedure.
- Building complex proof procedures from elementary proof tactics.

Proof-by-pointing [3] is a method for graphically constructing proofs. It includes the generation of proof scripts from user interactions, also called "proof script management". This approach could be used in order to perform general forward or backward reasoning on the premises and the conclusions of a goal. For example, in case of forward reasoning from one of the premises, this would entail first selecting the appropriate premise, then selecting the theorem which is used for the forward reasoning, and finally selecting the "forward reason tactic" that is to be carried out. Even simpler would be support for a "proof by clicking" of a subgoal by assumption — in this case the user would only have to select the premise that matches with the goal conclusion and then select the "assumption tactic".

As mentioned in the previous subsection, a naive implementation of proof-by-pointing might lead to brittle proofs. The root of this problem lies in the identification of terms via their position, for examples as the "third premise". More advanced implementations could instead use patterns which uniquely identify the selected term(s), for example as "the first premise of the form $(?a \subseteq ?b)$." In this way, it should be possible to generate proof scripts that are robust against small changes in the context or the statement of theorems.

Customisation

In order to improve usability, it is important that proof assistant users can adapt the UI to their own preferences and to changes in the proof assistant. This includes:

- The adaptation of menus, menu items, tool bars, keyboard short cuts etc. A simple example would be the addition of a tool bar button which generates a particularly frequently used proof command.
- The adaptation of formatting, both of proof assistant outputs as well as UI elements: fonts, font sizes, resizing of windows, pretty-printer settings, colouring, etc.

Such adaptations should be context sensitive in the sense that they depend on different UI modes such as top-level mode, theory mode or proof mode.

Adapting the system needs to be recognised as a normal activity that deserves proper documentation. While some systems such as XEmacs offer customisation dialogues, it can be easier to directly edit configuration files as long as the format of these files is easily understandable.

Customisation should be possible at different levels such as project level, theory level and proof level. Settings on a more specific level should override setting on more general levels, i.e. setting the visibility of brackets in the context of a particular proof should override the setting of this parameter on the theory level.

6 Architectural and System Design Issues

Paradigms

When developing software systems, it helps to have paradigms and metaphors that guide the design. From the user point-of-view, the overall system paradigm that we propose for proof assistant UIs is that of a *control centre* similar to modern UML/programming IDEs (integrated development environments). From the system design point-of-view, the suggestion is for a platform with pluggable components.

Basic Architecture

The user interface is the "front end" layer of the system architecture of a theorem proving environment. Its role is to permit the user an efficient control of the different functionalities offered by the system.

In many software systems, the user interface is written in the same programming language as the rest of the system. In this case, the lower layers provide interfaces consisting of functions that can be called from the UI in order to carry out user requests.

With most current proof assistants, the situation is different. Rather than providing an Application Programmers Interface (API) in some programming language, the proof assistants offer a text-based interface. This can be used directly by a user and yields a basic console ("command line") HCI. In order to obtain a more advanced user interface, one builds a separate, stand-alone UI which communicates via the text interface with the proof assistant. It is this architecture which will be assumed in the sequel by default. This is of course the architecture underlying systems such as ProofGeneral, PVS and PCoq.

Would it make sense to go even further and divide the UI into separate stand-alone components? For example, what about using a standard web browser for reading theories, a self-contained component for specifying theories and a separate interactive proof component for deriving new theorems?

- Using a standard web browser for reading theories is attractive because of reuse possibilities, hyperlinking and the widespread familiarity of users with this interface. MathML promises to simplify the rendering of mathematical formulas. Representing theories as XML documents would allow simple customisation of the presentation via XSLT or via a transformation to HTML combined with the use of CSS style sheets.
- Having two separate theory specification and proof derivation components would require dealing with consistency problems in case of concurrent access. In particular, the re-definition of constants could invalidate concurrent proof attempts that refer to these constants. There are also issues relating to the architecture of the proof assistant. For example, for the HOL family of provers, all components would need to talk to a single theorem prover process. Access to that process would have to be interleaved.

Proof Assistants as Components

According to the architecture suggested above, the proof assistant is not used directly via a console interface but indirectly via a user interface. This amounts to a significant change of perspective for some popular proof assistants. Traditionally, a system such as Isabelle has been viewed as complete in itself, rather than as being a component of a larger system. This change of paradigm has as number of implications for the proof assistant. For example, if proof scripts are generated automatically from a "proof by clicking", then it is no longer paramount that proof commands use a terse human-oriented syntax. Instead, one might want to distinguish between two languages - one for the communication between UI and proof assistant and another language for direct user input and the display of theories.

More generally, the separation of proof assistants from the rest of the system leads to a view of proof assistants as reactive components that provide a mathematical service in a distributed environment. This raises many questions, ranging from security issues and the support for concurrent access to the question of how to deal with a prover that does not respond due to an infinite loop in a proof attempt.

Allocation of Responsibilities

System design entails the allocation of responsibilities to different components. In case of a UI and a proof assistant backend, the assignment of most tasks is not a real question — the roles of the two components are clearly defined and quite separate. Still, some design decisions need to be made at the points where the two systems meet.

- For efficiency reasons, it is usually advisable that the UI performs basic validation of user inputs. It also makes sense for the UI to be aware of the textual commands that might be entered by users. This could help with graphical support as outlined in the HCI design section. On the other hand, parsing of formulas is a complex, logic-dependent task. It seems best if these are passed directly from the UI to the proof assistant. Having a parser in the proof assistant also supports the batch-processing of theory files.
- According to the design principle "keep data in its semantic presentation as long as possible", it is advisable for the proof assistant to leave most of the pretty printing to the UI. However, the proof assistant might perform some basic steps to bring the internal representation of terms and theories into a form acceptable by a generic interface.
- For efficiency reasons, one might consider to perform some session handling

tasks in the user interface, For example, by placing the proof history in the UI, one could reduce the amount of communication required for undoing a proof step as the new proof state could be retrieved within the UI. However, the simpler design is to keep these responsibilities with the proof assistant as it is the "expert" as far as the proof state and logical contexts are concerned.

Any final decisions about the allocation of responsibilities will also need to carefully consider the anticipated deployment scenarios and the performance characteristics of the different components and communication channels.

Component Frameworks and XML Protocols

Object-based component frameworks [14] such as CORBA, COM+ and Enterprise Java Beans (EJBs) support distributed computing. They rely on extensive runtime environments that hide many of the complexities of programming in a distributed and/or heterogeneous environment. In addition, these frameworks also provide a varying amount of support for other "middleware services" such as transactions, security, database pooling, pooling of object instances, etc. In these frameworks, components are associated with one or more interfaces consisting of typed methods that can be invoked from other components. The frameworks ensure type safety: the methods in a component implementation have to comply with the types (including possibly exceptions) in the interface declaration for that component.

The appropriate choice of component/distributed computing framework depends on many factors such as the amount of distribution and concurrency, performance requirements as well as the implementation languages and the target operating systems. Having said this, it would appear that because of the platform diversity in the academic community, a single operating system solution such as COM+ is unacceptable as a framework for building reusable UI and theorem prover components. It is also not clear that many of the facilities offered by CORBA, COM+ and EJBs would be useful with the current generation of proof assistants. Because of the non-negligible overhead, the chosen framework should be as simple as possible. In the light of these arguments it seems that simpler approaches are preferable. For platforms written in Java, the PCoq approach of encapsulating foreign language components as Java processes seems a natural framework. Components are accessed via Java APIs and distribution can be supported using either sockets or Java RMI.

Recently, XML message passing has appeared as an alternative approach to distributed computing. It has attracted attention in form of SOAP or XML-RPC web services that run via HTTP [7]. In this approach, components provide typically only a small number of interfaces over which data encoded as XML documents is exchanged. The validity of XML messages with respect to pre-defined rules is not ensured by the framework. Instead, validity can be checked explicitly by the receiver of a message. This results in very loosely coupled, easily extensible and lightweight architectures.

XML processing and web service standards are supported by implementations for different platforms and programming languages while at the same time being completely platform and programming language independent. A downside is that XML messaging is generally slower than the exchange of data via remote procedure calls or remote method invocations. Despite this performance drawback, XML messaging appears a promising choice for communication between generic UIs and proof assistant backends.

D. Aspinall has written a white paper [8] that suggests an XML vocabulary for communication between the major components in a theorem proving environment. Here are some general remarks concerning such XML-based protocols between UIs and proof assistants.

- The protocol needs to specify both the set of permissible XML messages (the "vocabulary") as well as rules about the permissible sequencing of messages. In particular, this includes the allowed request/response pairs as well as auxiliary messages that help with synchronisation.
- For the handling of commands and formulas in the XML protocol, one needs to take into account the placement of pretty-printer and parsing components as well as the need to support advanced graphical UI operations.
- For performance reasons, it is essential that the protocol also supports coarse-grained concurrency. This means that the proof assistant backend can process larger units in one chunk without the need for continuous synchronisation. For example, for the purpose of loading library theories, it is desirable to have XML markup for a "load theory XYZ" request such that no other messages are exchanged until the theory either is completely loaded or a failure occurs.
- The XML vocabulary should only regulate aspects which are directly relevant for the UI/prover communication and which are not covered by other standards such as MathML.
- The XML vocabulary needs to be planned with future extensions in mind.
- In order to further reuse, the XML protocol should contain a common core that is independent of proof assistants and user interfaces.

It is clear that the last point might well prove difficult in practice - getting different proof assistant and UI implementors to agree on a standardised core

XML vocabulary and message sequencing rules will not be easy.

Adaptability

According to the platform paradigm, the theorem proving environment features an extensible set of components such as theory editors, theory browsers, interactive proof components, MathML display engines, connectors to other provers, etc. According to the principles of component-based software engineering, these software parts should be easily reusable and customisable. In order to achieve this, one should aim for generic interfaces during the design. This should be accompanied by implementations that adhere to the principle of "programming to interfaces". The formulation of generic interfaces can benefit from considering different application scenarios. For example one might want to make it an explicit requirement that a proof display component supports two different provers.

In practice, a simple "plugging in" of more advanced components such as a tree editor can not be expected. Even if there were standardised interfaces for such components, a specific component implementation might well provide additional features requiring adapter code. However for such complex components, this is justified. The situation is different for simpler parts such as XSLT input/output filters. In this case, the platform should provide standard hooks for plugging in such a component. For instance, a program verification project could use a specialised filter that transforms programs from their usual syntax into an abstract syntax tree representation processed by the proof assistant. The addition of such a component to the platform should be possible without having to change program code in the platform.

For theory proving applications, the dynamic extension of a running system with new components does not seem necessary. Instead it seems sufficient to use configuration files that are loaded at startup time. This is the approach taken in the XUL [5] framework which underlies recent versions of the Mozilla web browser suite. As this project shows, it is quite feasible to extensively adapt user interfaces with the help of configuration files.

7 Concluding Remarks

Interactive proof assistants need to be accompanied by effective user interfaces. This is essential in order to make proof assistants accessible to a wider audience. It is hoped that the reflections on requirements and design in this paper will contribute to the achievement of this goal.

Despite the aim to stay generic, the discussion has doubtlessly been influenced by the author's experience with proof assistants from the HOL family and PVS. This bias should be rectified in future work by looking more closely at other automated reasoning systems and their user interfaces.

Many issues have not been touched on at all. This is particularly regrettable with regard to proof planning [6], multi-modal facilities [25], recent projects that use the web to distribute mathematical knowledge [17], and co-operative theorem proving [12,1].

Acknowledgement

The author would like to thank the anonymous reviewers for their detailed and constructive criticism.

References

[1] Allen, S., R. Constable, R. Eaton, C. Kreitz and L. Lorigo, *The Nuprl open logical environment*, in: *CADE: International Conference on Automated Deduction*, LNCS 1831 (2000), pp. 170–176.

[2] Amerkad, A., Y. Bertot, L. Rideau and L. Pottier, *Mathematics and proof presentation in Pcoq*, in: *Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, 2001. `http://www-sop.inria.fr/lemme/pcoq`

[3] Bertot, Y. and L. Thery, *A generic approach to building user interfaces for theorem provers*, Journal of Symbolic Computation **25** (1998), pp. 161–194.

[4] Booch, G., J. Rumbaugh and I. Jacobson, "The Unified Modeling Language User Guide," Addison-Wesley, 1999.

[5] Bullard, V., K. Smith and M. Daconta, "Essential XUL Programming," Wiley, 2001.

[6] Bundy, A., *A critique of proof planning*, in: A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, LNCS **2408** (2002), pp. 160–177.

[7] Cerami, E., "Web Services Essential," O'Reilly, 2002.

[8] David Aspinall, "Proof General Kit – White Paper (Version 1.4)," (2000). `http://proofgeneral.inf.ed.ac.uk/kit`

[9] David Aspinall and Thomas Kleymann, "Proof General User Manual – Version 3.4," (2002). `http://proofgeneral.inf.ed.ac.uk/`

[10] Dix, A., J. Finlay, G. Abowd and R. Beale, "Human-Computer Interaction," Prentice Hall, 1998.

[11] Eastaughffe, K., *Support for interactive theorem proving: Some design principles and their application*, in: R. Backhouse, editor, *User Interfaces for Theorem Provers (UITP'98)*, Computing Science Reports, Eindhoven, 1998 .

[12] Goguen, J., *Social and semiotic analyses for theorem prover user interface design*, Formal Aspects of Computing **11** (1999), pp. 272–301.

[13] Gordon, M. J. C. and T. F. Melham, editors, "Introduction to HOL: A theorem proving environment for higher order logic," Cambridge University Press, 1993.

[14] Heinemann, G. T. and W. T. Councill, "Component-Based Software Engineering: putting the pieces together," Addison-Wesley, 2001.

[15] Huisman, M., “Reasoning about Java Programs in higher order logic with PVS and Isabelle,” Ipa dissertation series, 2001-03, University of Nijmegen, Holland (2001).

[16] Jacobson, I., “Object-Oriented Software Engineering – A use-case driven approach,” Addison-Wesley, 1992.

[17] Kohlhase, M. and J. Zimmer, *System description: The MathWeb software bus for distributed mathematical reasoning*, in: A. Voronkov, editor, *Automated Deduction – CADE-18*, LNCS 2392 (2002), pp. 139–143.

[18] Larman, C., “Applying UML and Patterns,” Prentice Hall PTR, 2002.

[19] Lüth, C. and B. Wolff, *More about TAS and IsaWin: Tools for formal program development*, in: T. Maibaum, editor, *Fundamental Approaches to Software Engineering FASE 2000/ ETAPS 2000*, LNCS 1783 (2000), pp. 367–370.

[20] Lüth, C. and B. Wolff, *sml_tk: Functional programming for GUIs – reference manual*, Technical Report 158, Albert-Ludwigs-Universität Freiburg (2001).

[21] Merriam, N. and M. Harrison, *What is wrong with GUIs for theorem provers*, in: *User Interfaces for Theorem Provers (UTIP'97)*, 1997.

[22] Meyer, B., “Object-Oriented Software Construction,” Prentice-Hall, Englewood Cliffs, 1997, 2nd edition. http://archive.eiffel.com/doc/oosc/

[23] Nipkow, T., L. C. Paulson and M. Wenzel, “Isabelle/HOL — A Proof Assistant for Higher-Order Logic,” LNCS **2283**, Springer, 2002.

[24] S. Owre and N. Shankar and J.M. Rushby and D.W.J. Stringert-Calvert, “PVS System Guide – Version 2.4,” (2001). http://pvs.csl.sri.com

[25] Siekmann, J., S. M. Hess, C. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, M. Pollet and V. Sorge, *$\mathcal{LOUI}$: $\mathcal{L}$ovely Ω $\mathcal{U}$ser $\mathcal{I}$nterface*, Formal Aspects of Computing **11** (1999), pp. 326–342.

[26] Siekmann, J. et al., *Proof development with OMEGA*, in: A. Voronkov, editor, *Automated Deduction – CADE-18*, LNCS 2392 (2002), pp. 144–149.

[27] The Coq Development Team, “The Coq Proof Assistant Reference Manual – Version V7.4,” (2003). http://coq.inria.fr

[28] Wiedijk, F., *Comparing mathematical provers*, in: *Mathematical Knowledge Management, Proceedings of MKM 2003*, 2003, pp. 188–202.

[29] W.M. Farmer and J.D. Guttman and F.J. Thayer, “The IMPS User's Manual First Edition, Version 2,” (1995). http://imps.mcmaster.ca/

[30] Xavier Leroy, “The Objective Caml system release 3.06 – Documentation and user's manual,” (2003). http://www.ocaml.org