Formal Aspects of Computing (1999) 11: 225–243

Formal Aspects of Computing

The CtCoq System: Design and Architecture

Yves Bertot

INRIA Sophia Antipolis, Sophia Antipolis Cedex, France

Keywords: Interactive proof development; Graphical user interfaces; CtCoq; Proof by pointing; Proof maintenance

Abstract. The CtCoq user-interface is a graphical user-interface designed to be added to the Coq proof development system, acting as a broker between the human user and the logical engine. The principal design goal for CtCoq was to support large-scale proof development and we claim that this user-interface helps to increase the productivity of Coq users through powerful capabilities for elaborate mathematical notations, mouse interaction, and script management. In this paper, we review the user interface implementation to show how this design goal affects the capabilities provided by the system.

1. Introduction

Using formal methods to develop complex systems, designers and software developers create libraries of mathematical facts about the objects they consider. Formal mathematics, based on computer aided deduction systems should be used to check the consistency of the assumptions, or even sometimes, derive implementations from the specifications.

In [TBK92, BeT98] we claim that computer aided deduction systems need powerful user-interfaces and we lay out general techniques to construct such user-interfaces, based on a multi-process architecture and tools coming from programming environments. Following these ideas, we have developed a specific user-interface for the Coq system [DFH93], called CtCoq [BeB96], using the programming environment generator Centaur [BCD88]. Over the years, CtCoq has been used to experiment with new interaction ideas, such as proof-by-pointing [BKT94], drag-and-drop [Ber97a], selective undo [Pon97], and script management. We have accumulated a wealth of experience regarding the integration of

Correspondence and offprint requests to: Y. Bertot, INRIA Sophia Antipolis, 2004 route des lucioles, BP 93, 06902 Sophia Antipolis Cedex, France. Email: Yves.Bertot@sophia.inria.fr.

a variety of functions, their use in practical proof development and maintenance, and the ease with which these functions can be maintained as the context evolves.

The principal design goal was to support large-scale proof development. It is the main reason why we chose to adapt an interface to an existing theorem prover, instead of developing an interactive system from scratch. This choice imposed on us the constraint of accommodating the features of a proof system whose usage model was already influenced by a text-based interface. On the other hand, it made it possible to perform large-scale experiments [BeF95, Thé98, Ber98], based on the capabilities and theorem libraries already provided by the proof system. The user community of CtCoq is relatively modest (between 20 and 50 users down-load the user-interface at each release), owing to the fact that it only encompasses part of the Coq user community. It is hard to draw statistical conclusions from such a user base. Still, dedicated users both inside the developer's team and in industrial settings have been able to provide valuable feedback on the strengths and weaknesses of the tool. Thanks to our experiments and the feedback from these Coq users, we can claim that most of the ideas described in this paper have already been tested in realistic experiments. Keeping large-scale proof development in mind, it is also understandable that we paid less attention to beginners than in other user-interfaces, such as those aimed at educational purposes. We had to maintain a balance between the powerful functions we want to provide and the over-simplification that would be required to make a beginner feel comfortable. As a result, the CtCoq user-interface requires a longer training period than is usually required for a graphical user-interface.

In this paper, we want to review all the aspects of developing a tool like CtCoq and making it usable. Some of these aspects take their origin in ideas already present in [BeT98], but they have often been extended and adapted to integrate the lessons learned from the long-lasting experiment of maintaining CtCoq and collecting the reactions from users. This paper is structured as follows. We first review related work. Then we describe the architecture of the CtCoq system. Then, we review the characteristics of data display and extensible notations. In Section 5, we review the tools to guide proofs using the mouse. In Section 6, we review the tools for script management. In the final section, we draw conclusions and explain possible directions of research.

2. Related Work

Proof tools for which a significant effort has been put into the design of a graphical user-interface start with the Nuprl system [CAB86] and the Interactive Proof Editor [Rit88]. Both introduce a structure editor to maintain and edit proofs-in-progress. By comparison, we give less direct access to the proof structure, although we also use structure editing. Nuprl does not provide any provision for textual editing, while IPE relies on textual editing to enter formulas and then does not provide structure manipulation inside formulas. We should also note that IPE introduced the notion of proof-by-pointing, which was then refined in [BKT94] to make it recursive. To date, IPE is no longer used or maintained anymore, while Nuprl is still actively used and maintained.

The $\text{GLEF}_{\text{ATINF}}$ interface [CaH95] is close to CtCoq in its use of structure editing and pointing capabilities. From the design point of view, this interface was built at the same time as the logical engine, but efforts are being made to provide support for external proof engines. This interface relies very much on the

McIntosh model of interaction and we have been unable to ascertain whether its capabilities scale up very well. The logical framework ALF [MaN94] also relies on structural manipulation and is another case where the user-interface and the logical engine were developed simultaneously. As the model of usage relies on direct manipulation of proof objects (instead of using automatic or programmable tactics), there are problems treated in our work which are ignored in that system.

Other user-interfaces that have been designed as a complement to existing theorem provers are TkHOL [Sym95], Barnacle [LBM98], and XIsabelle [OEC97]. These systems rely mostly on a text-oriented graphical toolkit and are therefore less efficient in their use of the mouse to support formula manipulation.

More distant from our work are the experiments in the development of tools designed to teach basic logic. An interesting set of guidelines for such systems is described in [Sup84]. More recently educational systems have been listed in the review [GRB93]. Among these systems, it is interesting to note the Jape system [BoS94], because it uses direct manipulation features that are similar to the ones presented in this work.

Very few general man-machine interaction principles have been used in the design of CtCoq. Still, several features can *a posteriori* be seen as adhering to such principles. For instance, our work on proof-by-pointing follows the equal opportunity guideline advocated in [Thi90]. Also, our design agrees with [MeH97] in asserting that a "document" oriented approach makes it easier to reduce viscosity, although our documents are not plain text as mentioned in that work.

When compared with the papers [TBK92, BeT98, BKT94], this paper adds the perspective of several years of experience with using CtCoq for large-scale developments and maintaining the graphical user-interface through the evolution of Coq itself. From this perspective, we feel there is a need to review most of the capabilities provided by the interface and assess how these capabilities resist to the test of large-scale development.

3. Architecture

3.1. Basic Principles

Powerful notations are essential for making a usable mathematical tool. Mathematical notations are hard to obtain if the basic data structure is text, even when it is enriched with special characters as in [BoS94]. To make sure our design will make powerful notations possible, an early design choice was to organize all editing facilities around structure-oriented editing. In CtCoq, the basic data manipulated by the user is a tree-like structure, even though it appears on the screen as lines of characters. Gateways to textual forms have been provided, as will be seen in Section 3.2.1, but important commands require structured data, and any textual data is parsed before use.

The tree structured data is displayed on screen using a pretty-printing procedure. This procedure uses of a variety of fonts, size, and colors to render mathematical formulas. This capability is also provided in other interactive tools such as Mathspad [BVW97], which is also based on structured manipulation of mathematical formulas, using *stencils*.

Structure-directed editing is a debated feature: the constraints it imposes often appear as counter-productive and hard to learn, when one only considers

the task of entering basic commands. For this reason, much effort has been devoted to integrating smoothly text editing in the user-interface. Still, structured data provides opportunities for more powerful manipulation tools, as has already been shown in the facilities for mouse-directed proof (see Section 5). We also hope to use these facilities to develop powerful proof maintenance tools.

3.2. Multi-processing Aspects

As advocated in [BeT98], CtCoq communicates with Coq using TCP sockets, using a different protocol in each direction. Commands sent to Coq are sent as plain text in Coq's input syntax. Data coming back from Coq is transferred as a tree-like structure, using the protocols of [DéR94] to encode data.

For large scale proof development, it even proved worth the effort to implement a distributed approach, using different machines to run the various processes: in this setting the two processes do not compete for cycles and memory on the same machine. This significantly improves response time as it avoids the overhead of the operating system swapping context between the two processes.

To make this approach maintainable and adapted to large-scale proof development we have had to consider problems around extensible parsing, protocol procedures, update efficiency, and possibilities to interrupt the logical engine.

3.2.1. Parsing and Logical Processing

Although the only first class data in the interface is tree-like structures, CtCoq also provides ways to input textual data. For this, a parser is required. Developing a parser for Coq commands from scratch or from a separate abstract description raises maintenance problems, as it is necessary to check that the parser works like the one in the Coq prover.

Instead we used the parser from Coq. Coq is developed in `Ocaml` [ReV97], a dialect from the ML family [MTH90]. There is a central data structure called `CoqAst.t` that is used as a pivot for communication to and from the user. To communicate with the user-interface process, it is enough to implement a translator from this `CoqAst.t` structure to the tree structure used in the user-interface. In spite of the size of this translator, this has proved more practical than maintaining a different parser.

Instead of using the Coq system to handle parsing requests as well as logical processing requests, we have reverse engineered the Coq source code to extract a smaller program that only takes care of the parsing. Thus, the CtCoq user-interface can handle parsing requests even if the Coq process is busy performing a logical processing request.

The Coq system also provides extensible syntax, in the sense that users can provide new grammar rules that are merged into the parser. Due to this capability, the grammar which describes syntactically correct expressions is no longer context-free: sentences that appear after a grammar rule in the text are supposed to be parsed using this grammar rule (among others), while sentences that appear before are supposed to be parsed without. Enforcing this behavior is simple in bare Coq, since a document's contents are always parsed in order. In the CtCoq interface, the usage model we have is that the file is a document in which users navigate freely, so that they may edit fragments in any order.

Sentences that appear before the added grammar rule may need to be parsed after the grammar rule has been added to the parser.

The CtCoq system accepts syntax extensions but all data is parsed with the current configuration of the parser. This may lead to inconsistencies, but no problems will occur as long as entire proof developments are done using the same syntax. Syntax extensions may be loaded once and for all at the beginning of the session. We provide ways for the user to set up configuration files so that their preferred syntax extensions are automatically loaded when the parser starts. We also make it possible to restart the parser from scratch in the middle of a session. As the user may change the configuration file between different restarts, this gives a poor man's capability for variable syntax.

3.2.2. Protocol Production in the Coq System

The Ocaml type-checker has been used to make a more robust implementation of the communication protocol.

This protocol uses a textual representation of tree-like structures based on a postfix traversal of the structures [DéR94]. On the emitting side, the data to transmit is in the Ocaml data-type `CoqAst.t`. This data-type actually provides a very simple notion of trees: it has few operators: `Node` (with a string and a list of `CoqAst.t` terms as arguments) or `Var` (with a string as argument), for instance. The `CoqAst.t` data-type does not enforce any form of syntax discipline.

On the CtCoq side, the text manipulated by the user must correspond to a set of constraints called an *abstract syntax*. It is described using 350 operators and as many syntactic categories. These operators make it possible for the structure-editing tools to ensure that user will only construct syntactically legal commands. For instance, Coq provides the possibility to write function applications with two parts: a function and a non-empty list of arguments. In CtCoq, this construct is represented by an operator `app` with two fields. The first one must belong to the category `FORMULA` and the second one must belong to the category `NE_FORMULA_LIST`.

For the communication protocol, we must ensure that the text emitted by the Coq process or the parser process corresponds to a valid tree with respect to the abstract syntax constraints. This kind of program is hard to write, hard to debug, and hard to maintain.

A solution, proposed and implemented by Healfdene Goguen, is to develop a two-pass approach that uses mechanical verification in the first pass and mechanical code generation in the second pass. We produce automatically an intermediate Ocaml data-type that provides a strongly typed description of the Coq system syntax from the abstract syntax description used in CtCoq. This description is a large set of mutually recursive types corresponding to the syntactic categories used in CtCoq, where the constructors correspond to the operators used in CtCoq. The correspondance is not completely direct, because some abstract syntax operators may belong to several syntactic categories while Ocaml constructors only belong to one type. The tool also produces functions mapping terms in the intermediate type to valid strings for the communication protocol. These functions are correct by construction. Now, we only need a program that maps `CoqAst.t` terms to terms in the intermediary Ocaml type. This program must be written by hand and needs to be updated when Coq evolves, but the Ocaml type-checker comes in to help detecting errors.

For instance, the mathematical expression $x + y$ will be represented inside Coq by the following expression:

```
Node("APPLIST", [Var "plus", Var "x", Var "y"])
```

The CtCoq abstract syntax contains the following declarations:

```
FORMULA ::= app ID lambda ...;      ID ::= id;
app -> FORMULA FORMULA_NE_LIST;
```

Our tool generates the following type declarations and functions:

```
type fORMULA =
  App of fORMULA * f_NE_LIST | Coerce_ID_to_FORMULA of iD ...
and iD = Id of string and ...

let rec fORMULA_to_string = function
  App(a,b) -> (fORMULA_to_string a)^(f_NE_LIST_to_string b) ^
    "n\n" ^ "app\n" ^ "2\n"
| Coerce_ID_to_FORMULA a -> (iD_to_string a) ...
```

Only the following function needs to be written by hand in Ocaml:

```
let rec (f: CoqAst.t -> fORMULA) = function
  Node("APPLIST",op::a1::args) ->
    App(f op, F_ne_list(a1 op,  map f args))
| Var(x) -> Coerce_ID_to_FORMULA(Id(x)) ...
```

Computing

```
fORMULA_TO_STRING(
    f(Node("APPLIST", [Var "plus", Var "x", Var "y"])))
```

returns (where line feeds have been replaced by horizontal space):

```
"a id plus a id x a id y n formula_ne_list 2 n app 2 "
```

This is the correct protocol string for $x + y$.

Implementing this approach has been a real progress in the design of CtCoq. The hand-written part remains tricky to maintain, since we do not have a declarative description of the `CoqAst.t` terms that may occur in a session. Until now, we have relied on reverse-engineering in the Coq system to infer this information.

3.2.3. State Update

The CtCoq user-interface must display a clear view of the Coq process's state. As a result, data must be duplicated between the logical engine and the user-interface. The naive solution is to have the logical engine send a complete description of its state after each interaction, but this is inefficient.

We observed that a complete view of the Coq process' state is not necessary all the time. For instance, proof maintenance activities often require replaying long fragments of scripts with a low error rate. In this case, it is not necessary to have the intermediary states displayed: the state should only be updated at the end of execution or when an error occurs.

Taking these observations into account, we have designed a protocol where

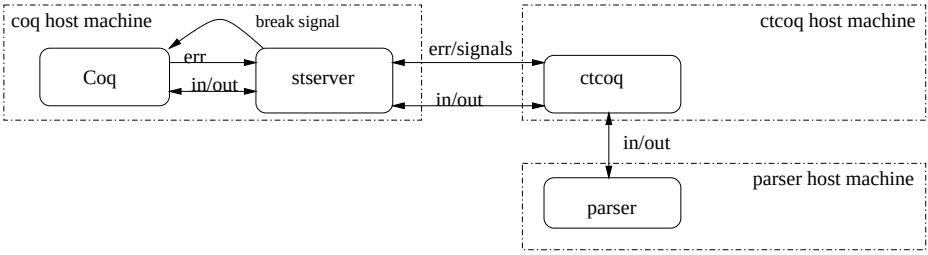

Fig. 1. The multi-process architecture of CtCoq. An auxiliary `stserver` process takes care of interpreting "signal" messages from the user-interface as Unix `kill` commands applied on the Coq process. In regular usage, all processes run on the same machine.

the logical engine only sends a summary of its state to the user-interface after executing each command. In return, the user-interface computes the data it actually needs and requests it from the logical engine. This protocol implements two modes of communication: *replay* and *regular*. In *regular* mode, the user-interface receives the information summary, computes the goal it wants to display and requests only this goal from the logical engine. In *replay* mode, the user-interface simply marks that the current command has been executed and sends the next command. If an error occurs while in *replay* mode, execution is stopped, the mode is changed back to *regular*, and the state is updated.

3.2.4. Interrupting the Prover

Some of the commands available in the logical engine may provoke lengthy or looping computations. The user-interface is not complete if it does not provide a way to interrupt these computations. In CtCoq, it is provided by an option in a menu and its behavior relies on sockets and Unix signals.

Since the two processes may not run on the same machine, it is difficult for the CtCoq process to find the Coq process identification before sending it a signal. We solved this problem by adapting the tool encapsulation tools described in [Cle90] and called *stservers* in [JMB93]. A small process called a `stserver` is added between the user-interface and the logical engine. This process acts as a broker. It runs on the same machine as the logical engine, communicates with it through its standard input, output, and error character streams, and may occasionally send a signal. On the other side, the `stserver` process communicates with the user-interface process using two bi-directional sockets (see Fig. 1). One socket is used to transmit the standard input and output character streams, the other socket is used to transmit the standard error stream (from `stserver` to CtCoq) and the fourth direction is used to send "signal" messages from CtCoq to the `stserver`. During normal use, the `stserver` process only copies from one character stream to the other. When receiving a message on the signal line from CtCoq, it does not copy it, but sends a signal to the Coq process, whose process identifier is known since creation time.

We also have to make sure that the Coq process returns to a reasonable state after receiving an interrupt signal. Messages sent by Coq to CtCoq all have a regular form, following the encapsulation proposed in [TBK92]. Messages have a header (used on the receiving end to decide how to parse the contents), some content, and an end marker. If the Coq process is interrupted between the moment when it outputs the header and the end marker, the receiving-end

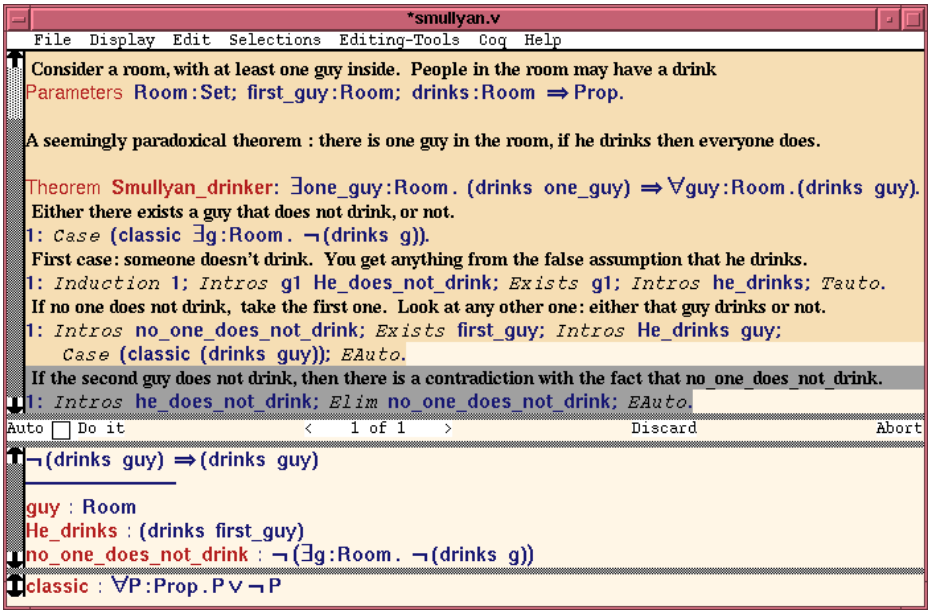

Fig. 2. A composite window. The top area contains the script. Different backgrounds are used to distinguish commands that have already been executed and the "current" command. The middle area contains the current goal. The text above the short horizontal bar ($\neg(drinks\ guy) \Rightarrow (drinks\ guy)$) is the expression to prove, the text below the horizontal bar ($guy : Room \ldots$) is a local context of named known facts. The bottom area lists theorems when the user requests it. The relative size of sub-windows can be adjusted using the mouse.

protocol procedures get stuck, expecting an end marker that never comes. The solution is to protect carefully the data emitting sections with a signal handling function, which will memorize the fact that a signal was received instead of brutally interrupting the computation. Of course, the protection should only be transient and received signals should be processed when exiting the protected section.

3.3. Screen Real Estate

From the beginning, we have designed interfaces that provide many windows, each adapted to displaying one kind of data or performing one kind of activity. It is important to avoid having too many windows on the screen. A good solution to this is the composite window already presented in [BeT98] and shown in Fig. 2.

In a large proof development, users may reach situations where they feel the need for a new lemma in the middle of a large proof. Productivity increases if users are able to open a new session with the Coq process, working practically in the same context, to prove the missing lemma and then resume the initial proof. The CtCoq user-interface provides this capability: several composite windows may be open at the same time, each one hosting a virtual session with the Coq system.

Users have the possibility to work simultaneously with several virtual sessions, sending commands arbitrarily from several composite windows. It is important

to make sure the right composite window receives the results. For this, every composite window is given a unique identifier. The commands sent to the proof engine are encapsulated with a header that also gives this identifier to the proof engine. The commands are processed in the logical engine in a regular fashion but results are also encapsulated with a header that gives the window identifier. Inside the user-interface process, the message is then routed back to the right composite window.

3.4. Safety

Our design helps to safeguard scripts against erroneous manipulations, using three features.

The first feature is to separate the script window into several areas, where the area containing the commands that have already been executed is read-only. Enforcing this constraint is relevant because editing commands and sending these commands to the logical engine are two different steps and because the executed commands and the yet-to-be-executed commands are stored in the same editing area. We assume this question is meaningless in systems like ALF [MaN94] or Jape [BoS94] where editing and proving are merged in one single activity.

One conventional way to help protect precious data is to have the user-interface save regular backup files on disk. We have also implemented this capability, by including an object in our development that schedules regular alarms and sends a message to all composite windows to save the contents of their script sub-window. In this case, the scripts are saved in tree form rather than in regular text form, so that commands that are incomplete when the backup is performed can still be reloaded.

We also went one step further by setting up a memory safety mechanism. This safety provokes the termination of the user-interface when the process runs short of memory. When this condition is detected, a memory reserve is freed, the backup mechanism is triggered (assuming there is more room in memory), and the processes are killed. This safety mechanism works by checking the statistics of the garbage collector. It also provides a warning mechanism that will tell the user when memory reserves are getting low, but much sooner than when the more drastic termination procedure is triggered.

4. Display

The tree-like structures manipulated in CtCoq are transformed into text using three different procedures, to save in files, to communicate with the Coq process, or to display on the screen. The notations used on screen do not need to be the same as the ones used for storage and for communication with Coq. For storage and communication with Coq, the text needs to be unambiguous but it does not matter whether the notations used are concise or elegant. On the other hand, the data displayed on screen should be concise and elegant to help readability and some amount of ambiguity is acceptable.

In mathematics, new notations are customarily introduced to handle specific domains. For this reason, it is also interesting to let users of CtCoq adapt the notations that appear on screen. The Centaur programming environment generator [BCD88], on which CtCoq is built, provides an abstract language,

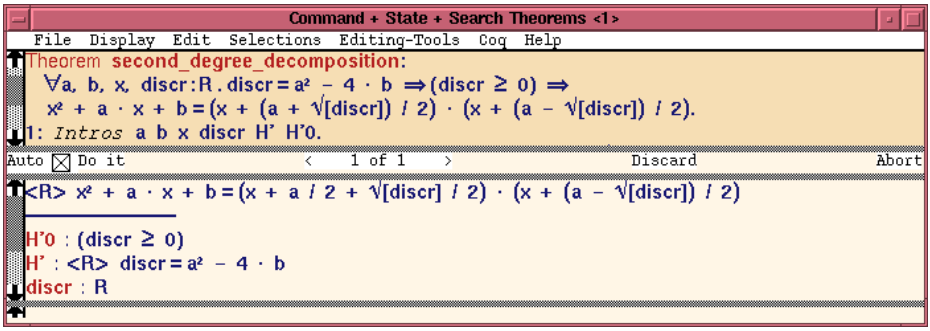

Fig. 3. Notations for real number algebraic calculus.

called PPML (Pretty Printing Meta Language), to describe the layout [MCC86]. This layout mechanism makes it possible to use a variety of fonts and colors, as could already be seen in Fig. 2.

The layout mechanism can be configured by the user in several ways. First, users can change colors and fonts at will. Each character is displayed with a class, and the background color, foreground color, and font associated to each class can be tuned to the user's taste.

The second way to configure layout is to add pretty printing rules to the description. This is possible because Centaur provides a mechanism of layered pretty printers: the layout is not controlled by one specification file, but by several specifications placed in a list. Each specification file contains a collection of rules, each mapping a tree pattern to a layout pattern. For a given node in the displayed tree, if no tree pattern from the first specification matches, then the layout mechanism looks up in the second specification, and so on. Different lists of specifications are used for printing in files and for displaying on the screen.

Usually, the first element in the specification file list is a small file that describes the user's preferences. Figure 3 shows an example of user-prescribed notations. The layout mechanism also provides postscript output that makes it possible to produce insertions to include into paper descriptions of the mathematical developments performed using CtCoq.

The PPML machinery provides only text-like layout, with an abstraction based on layout boxes to perform indentation. The examples given in 3 may mislead the reader into believing that this is sufficient for general mathematical notations. It is not. The exponentiation examples in Fig. 3 are only possible because one of the fonts available in the window system contains a character that represents the number 2 in exponent position. Exponents other than 1, 2, and 3 are not available. Similarly, subscripts are simply obtained by using a smaller font.

With these easily adaptable display mechanisms, CtCoq also provides the possibility to have several views of the same data at the same time, with a different layout specification for each view. Thus, it is possible to open an auxiliary "table of contents" window, that displays only the definitions and theorem statements, but not the proof commands.

Other authors have proposed enhancements to the PPML formalism, to make it easier to produce output data that hides the actual structure of the displayed tree [Bou94]. However, such enhancements often miss the point as they ignore the need for incrementality that is crucial for use in an interactive tool.

5. Mouse Directed Proof

The features of CtCoq that are most appreciated by users revolve around its capability to let the user guide proofs efficiently using the mouse. Of course, one could consider providing menus with all the Coq commands, but this would be much less efficient than text interaction. Instead, we have implemented proof-by-pointing [BKT94, BeT98] and drag-and-drop [Ber97a].

The first facility interprets clicking actions as a language to perform proofs in predicate calculus. For instance, let us consider the following goal:

$$\forall x : nat.\ (\exists y : nat.\ x = 2 \times y) \Rightarrow even(x).$$

If the user simply clicks on the second x (counting the one just after the quantifier) and chooses the option "rewrite" in the proof-by-pointing menu, then the gesture is interpreted as a command that will:

1. introduce a constant x on which to reason,
2. introduce hypothesis $\exists y : nat.\ x = 2 \times y$,
3. introduce a constant y and a hypothesis $x = 2 \times y$,
4. produce the goal $(even\ (2 \times y))$ where x has been rewritten according to the last introduced hypothesis.

The second facility interprets dragging the mouse as a language to perform rewriting. For instance, let us consider the following goal:

$$x + ((a + b) + z).$$

Dragging the mouse from x to a will provoke a rewriting operation that permutes x and $a + b$, leading to the following expression:

$$(a + b) + (x + z).$$

In the next two sections, we study separately the implementations of these facilities in CtCoq. The important idea in the first section is that the proof-by-pointing algorithm is a compiler from gesture to proof commands and that this compiler needs an optimization phase. The important idea in the second section is that tools to interpret mouse gestures can be extensible, with various levels of automation in the extension process.

5.1. Implementation Aspects of Proof-by-pointing

Precise descriptions of the proof-by-pointing algorithm can be found in [BKT94, BeT98]. Let us only summarize the main characteristics of this algorithm. Proof-by-pointing is used when the user clicks with the mouse on a sub-expression of a goal and asks for a command to be generated. The algorithm receives as data the complete goal as a tree and the path from its root to the branch that was selected. Computation then proceeds recursively, going down the tree structure along the path. At each step, an elementary command is generated using only information on the tree operator that is traversed (by pattern matching) and the next elementary step in the path. All the commands are chained together to form a large composite command that will create new goals when executed.

In [BKT94, BeT98], we show that the algorithm for proof-by-pointing can be implemented in the logical engine or the user-interface. If the processing is

performed in the logical engine, then the command that is sent to the theorem prover only needs to receive as argument a description of the path, since that process already holds a description of the goal. The same command should be recorded in the script. A drawback is that the path is meaningless without the goal and the script becomes hard to understand. This matters a lot when considering user-friendliness and script maintenance.

In CtCoq, we have implemented the algorithm inside the user-interface and made it use plain Coq commands. The proof-by-pointing algorithm generates a large composite command that is sent to the script window. In this respect, it works as if the state window, where the goal is displayed, was simply used as a "structured" menu where the user can click to choose a command that will be inserted in the proof window. Proof-by-pointing then becomes an editing command like any other. Thus, proof-by-pointing generated commands can be edited by the user before being sent to the logical engine. Manual editing is actually often required, because generated commands contain holes that need to be filled in before the command is sent. As a result, at least two mouse clicks are needed for every proof-by-pointing action: one to generate the command and the other to send this command to the logical engine. To avoid this annoying extra click, we also provide the possibility to set a flag that expresses that commands generated by proof-by-pointing can be sent immediately to the logical engine when they contain no holes. This capability is provided by the little check box that appears between "Auto" and "Do it", unchecked in Fig. 2, and checked in Fig. 3.

The fact that proof-by-pointing commands need to be edited by users is a weakness, as users need to understand their structure to fill in the holes. This weakness comes from the lack of higher-order unification capabilities in the logical engine. However, the holes are named in a way that users can understand the role of each requested piece of data.

The proof-by-pointing facility helps make proof scripts that are easier to maintain. In particular, it makes sure that explicit names are given to all created hypotheses, instead of relying on an automatic naming scheme. Having explicit names in the scripts helps by provoking the early detection of name clashes when proofs are adapted to different contexts.

It is important to make proof-by-pointing generated commands readable. For this reason, we devised a command simplifier [Ber97b]. The proof-by-pointing algorithm acts as a compiler that transforms symbolic descriptions of the user's movements into commands in another "programming" language. The simplifier is the optimizer of this compiler.

The output from the first phase of the proof-by-pointing algorithm, as it is described in [BKT94, BeT98] is a composite command that follows the path from the root of the goal down to the selected expression. The job of the simplifier is to look at command subsequences and recognize opportunities for compaction, often replacing sequences of elementary steps by one step that performs approximately the same operation. Having only an approximation of the original commands is acceptable as long as the main intent of the gesture is respected.

The algorithm works by using several traversals of the composite command from top to bottom, each traversal checking whether some collection of rewriting rules apply.

For instance, one of the first passes applies a rule for compacting introduction commands (commands that make it possible to go from a goal $A \Rightarrow B$ to a goal B with A as a known fact).

$$\texttt{Intros } x_1 \ \ldots \ x_n\texttt{; Intro } x_{n+1} \rightarrow \texttt{Intros } x_1 \ \ldots \ x_{n+1}.$$

When designing this simplifier it becomes apparent that a carefully designed command language makes user-interface construction easier. The more uniform the language is, the easier it is to provide automatic tools to manipulate composite commands. In the case of Coq, we have shown that many commands handle premises of implications in a uniform way: these commands are called *head tactics* in [Ber97b].

5.2. Drag-and-Drop Rewrite

The proof-by-pointing algorithm only takes care of predicate calculus. A lot of mathematical reasoning is more algebraic in nature, using general equations to replace expressions by equivalent ones. Users change the shape of the formula to prove until a simple form is recognized. We observed that many algebraic operations actually are used to combine data or move data around. From this observation stemmed basic notions for an interpretation of drag-and-drop movements with a simple implementation.

5.2.1. Classes of Equations

Many equations can be given an operational intuitive meaning, stating that some symbol moves around or that two symbols combine to produce a new result. Let us enumerate a collection of possible equation forms.

1. $x + y = y + x$ or $x + (y + z) = y + (x + z)$. These equations permute two expressions.
2. $x + (y + z) = (x + y) + z$ or $x \times (y + z) = x \times y + x \times z$. These equations rearrange the relative position of two operators.
3. $-(-x) = x$. This equation combines two operators.
4. $-(1/x) = 1/(-x)$. This equation permutes two operators.
5. $x - x = 0$. This equation combines two expressions.

When formulas are displayed on the screen, it is possible to convey the same operational intuition (rewriting left-to-right) with only a gesture of the mouse. In case (5), users only need to select one of the x's and drop it on the other. In case (1), they need to select y and drop it on x. In cases (3) or (4), they need to select the minus sign and drop it on the other operator. In case (2), they need to select the parentheses and move them to the other $+$ sign for the first equation, and select the x and move-it to the y or z for the second equation. Note that the reverse operation can also be expressed using a gesture for cases (1), (2), or (4).

5.2.2. Implementing Drag-and-Drop

The drag-and-drop engine takes three pieces of data as input. The first one is the goal expression, as a tree T. The second piece of data is a pair of paths p_1, p_2 in this tree to indicate the first point and last point of the user's gesture. The third piece of data is an ordered list of drag-and-drop rules, where each rule combines a pattern P, a pair of paths p_f, p_l, and an incomplete command C. The output is a command that is ready for the logical engine.

The algorithm first computes a third path p_3 from the pair p_1, p_2: the longest

common prefix to the two paths, this path points to the smallest expression that contains both the first point and the last point. Then for each rule, the algorithm performs the following steps:

1. Compute the longest common prefix p_r to the paths in the rule.
2. Compute the path p such that $p_3 = p \cdot p_r$, when such a path exists.
3. Check that $p_1 = p \cdot p_f$ and $p_2 = p \cdot p_l$.
4. Check that the sub-term of T at path p is an instance of the pattern P for some substitution σ.
5. Apply σ to C.

Steps 2, 3, or 4 may fail. In that case, the algorithm skips to the next rule. Step 1 may also be pre-compiled and stored in the rule description.

Obviously, the order of rules matters. If two rules (P, p_f, p_l, C) and (P', p'_f, p'_l, C') are such that there exist a substitution θ and two paths q and r so that the following equations hold

$$P' = \theta(P) \qquad p'_f = p_f \cdot q \qquad p'_l = p_l \cdot r$$

then (P', p'_f, p'_l, C') will never be used if it occurs behind (P, p_f, p_l, C).

To make the generated command obey precisely the user's gesture, the drag-and-drop algorithm also inserts a restricting command to express exactly where to rewrite, when several locations are possible.

5.3. User-level Extensibility

We have implemented tools to make the drag-and-drop engine easily extensible. Two directions have been studied, more or less based on automation.

There are several classes of theorems for which one can introduce drag-and-drop rules with systematic shapes. We have developed a procedure that analyzes the statement of theorems and constructs the corresponding drag-and-drop rules, when these theorems fall in the pre-defined classes. The various classes correspond to the theorem forms described in Section 5.2.1. For each theorem, the procedure will typically construct up to four rules: the equation can be oriented as a rewrite rule in two possible directions and for each case the gesture can itself be from left to right or from right to left. The idea of classifying rewrite rules according to their syntactic form is also present in work to guide automatic proof search [BuW81, BSH93]. Interestingly, some of the categories of rules described in these papers also carry a graphical intuition.

It is also possible to add rules manually, using a specialized editing tool to manipulate the pattern, the theorem, and the two paths (see Fig. 4). The rule editor is designed so that adding a new drag-and-drop rule is like programming by example. To enter the pattern, users only need to get an instance of this pattern from the other CtCoq windows using copy and paste actions. To enter the command, they only need to type an instance of this command. To enter the two paths, they give an example of the movement by dragging the mouse in the pattern window. The pattern and command must then be "generalized" by inserting "meta-variables" in the positions where arbitrary data may occur.

For the rule list, there is a "sort" command to make sure that general rules do not hide specialized ones. The user can also manually change the place of a rule by dragging it to the right place.

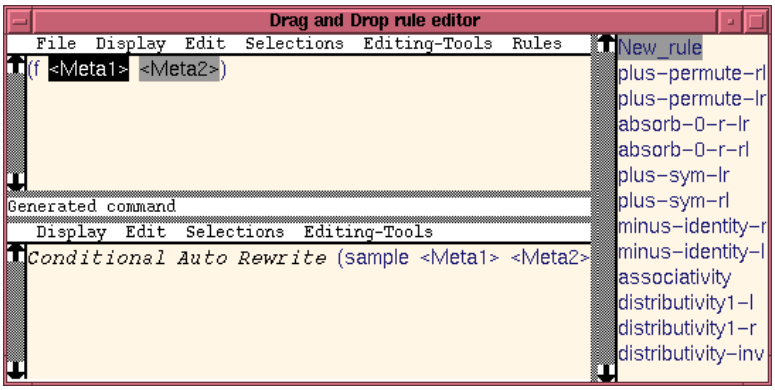

Fig. 4. The drag-and-drop rule editor. The top left window contains the formula pattern and indicates the start and end positions. The bottom left window contains the triggered rewrite command. The right window shows the ordered list of rules.

6. Script Management

The script management support in CtCoq encompasses a collection of commands to help visualize the structure of proof developments, return to past states, and keep coherent scripts. We define an insertion point in the script window, where executed commands must be inserted. This ensures that executed commands are recorded in the exact order of execution. The machinery makes sure that the contents of the script window down to the insertion point represent the current state of the proof engine.

The script management facility as described in [BeT98] is a broker between the user and the proof engine, updating the list of executed commands as the process goes on. In CtCoq, departures from the basic principle have had to be implemented to handle specific aspects of Coq and CtCoq: varieties of undo commands and the interaction between multiple windows and script management.

6.1. Varieties of Undo Commands

In the original design of [BeT98] undo commands only go backward in history in a relative manner: the `Undo` command undoes the last command, while the `Abort` command goes back to the last "start" command. We have also integrated a `Reset` command that refers to the point of return in an absolute manner. Given a theorem name, the `Reset` command undoes all the commands that occurred after the proof of this theorem, this proof included. The dialog with the proof engine is simple, as only the `Reset` command is sent. On the user-interface side, it is important to move the insertion point back to the correct place.

Another variety of `Undo` command that has been integrated in the user-interface is *logical* undo [Pon97]. This command is available when the user-interface has enough information about the actual dependencies between all the commands that were executed after the undone command. It makes it possible to undo only the commands that depend on the undone command, while keeping the others in the executed part of the script.

Chronological undo can be performed without moving data around in the

proof script. Only the insertion point needs to be moved from the current position to the undone position. As a result, it is possible to "redo" at a relatively low cost by simply replaying the undone fragment (for the user this will require between one and three mouse operations, plus waiting while the commands are replayed). With logical undo, some data needs to move around, to avoid undone commands being kept between *executed* commands. Our logical undo algorithm takes care of rearranging the undone commands so that they can be redone easily.

6.2. Script Management with Multiple Windows

Script management also has strong interaction with the multiple window capability of CtCoq. If the user can open two windows and send commands from these two windows in an arbitrary order then there is no way to ensure that any of these windows will contain a proof script that can be stored in a file and replayed independently. To take this into account, the user-interface keeps a hidden record of the order in which all commands have been executed. This record is used to help the user move data from one window to the other, without disturbing the actual order.

For instance, if the user has two composite windows 1 and 2, where window 1 contains the commands A, C, E, window 2 contains the commands B, D, F, and that the commands have been sent in alphabetical order, our tool makes it possible to move B from window 2 to window 1 and it enforces the constraint that B must be inserted between A and C.

Our tool could be more stringent, for instance by enforcing that saved files must contain contiguous segments of the master record. A quick reflection on this constraint shows that it would be counter-productive: when users develop results based on facts from two domains, for instance polynomials and real numbers, they may want to have in one window their lemmas about polynomials and in the other window the lemmas about real numbers, independently of the order in which these lemmas were proved.

7. Conclusion

CtCoq has been a sustained development effort over several years and the effort pays off in a really efficient user-interface for expert users. After several large scale experiments, we are confident that it presents no major design flaws. This experiment is a relevant inspiration for researchers interested in integrating a logical engine inside a working environment for formal methods.

There are three characteristic features in the CtCoq interface. The first one is its reliance on structure manipulation. This design choice is necessary to provide good mathematical notations, which are important to make formal methods accessible to a wide community of users. It has proven to be an opportunity, as it facilitated experimenting with new interaction capabilities. But from the user's point of view, structural manipulation is a curse for beginners and a necessary evil for expert users.

The second characteristic of CtCoq is its design as a complementary user-interface to an existing proof system that has its own life cycle. This choice makes it easier to attract new users and makes the goal of supporting large scale proof development more realistic, as developers engaged in such tasks can rely on the

mathematical libraries already provided by regular Coq users. This choice also has implications in design decisions: most of the time we have tried to minimize the amount of modification needed in the Coq system to implement the capabilities we envisioned, to make them less sensitive to evolutions of the Coq system. It often appears that some capabilities are implemented in the user-interface while it would seem natural to implement them in the logical engine. The seemingly unnatural choice is often guided by maintenance considerations.

The third characteristic of CtCoq is its use of pointing to support logical and mathematical manipulations. In this respect, the user-interface makes it possible to imagine new ways of interacting with the logical engine. So far, we have postponed work on these new modes of interaction, concentrating rather on facilities that could be based on the text-based commands of Coq, like the proof-by-pointing facility. Future work on CtCoq should contain more and more capabilities that would be impossible to provide in a text-oriented interface.

For most design decisions, we were faced with a number of trade-offs. As a result, the CtCoq user-interface also exhibits strong weaknesses. First, the user-interface is hard to learn for beginners, mostly because of its attachment to structural manipulation. Structural manipulation as it is performed in CtCoq is a constrained activity, where simple operations often require sequences of commands that are longer than expected by beginners. A drastic solution would be to abandon structural editing altogether, but this would also imply abandoning powerful mathematical notations. Rather, we hope to reduce the constraints of structural manipulation by adding recovery mechanisms that perform the relevant actions when the strict model of structured editing fails. Second, the multiple-process architecture is difficult to install and many potential users are rebuked by installation problems. Combined with the hard learning curve, this reduces the number of our potential users and makes user feedback harder to obtain. We hope to solve this weakness by coupling the installation processes of Coq and CtCoq and by porting CtCoq outside of the Centaur system. Another approach to coping with these weaknesses has been to understand how the good features of CtCoq, like proof-by-pointing, could be implemented in environments where some aspects are different [BKS97].

With large-scale proof development in mind, we have learned that proof maintenance is an important activity in the proof process. Proof maintenance concerns the actions taken by users when they want to adapt already proven theorems to different contexts: when an axiom is removed or modified. For large-scale development, proof maintenance activities appear in the proof life-cycle even before the complete development is over: as formalization progresses, users discover that some initial assumptions have to be re-considered, refined, or removed, and theorems that were proven with these assumptions need to be adapted. Recent research around CtCoq has been studying this aspect [PBR98].

Acknowledgements

I would like to thank Gilles Kahn and Laurent Théry, Janet Bertot, Francis Montagnac, Laurence Rideau, Healfdene Goguen, Yann Coscoy, and Olivier Pons for their help in the system development, The Coq team for their support in accommodating the needs of our interface, and the CtCoq users themselves, especially Jean-Franois Monin from CNET and Emmanuel Ledinot from Dassault Aviation, who helped both with ideas and funding.

References

[BeB96] Bertot, J. and Bertot, Y.: The ctcoq experience. In *Electronic proceedings for the conference UITP'96*, University of York, July 1996.

[BCD88] Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. and Pascual, V.: Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).

[Ber97a] Bertot, Y.: Direct manipulation of algebraic formulae in interactive proof systems. In *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, September 1997.

[Ber97b] Bertot, Y.: Head-tactics simplification. In *Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[Ber98] Bertot, Y.: A certified compiler for an imperative language. Research Report RR-3488, INRIA, 1998.

[BeF95] Bertot, Y. and Fraer, R.: Reasoning with Executable Specifications. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95)*, volume 915 of *Lecture Notes in Computer Science*, pages 531–545, 1995.

[BKS97] Bertot, Y., Kleymann-Schreiber, T. and Sequeira, D.: Implementing proof by pointing without a structure editor. Technical Report ECS-LFCS-97-368, University of Edinburgh, 1997. also available as INRIA Report RR-3286.

[BKT94] Bertot, Y., Kahn, G. and Théry, L.: Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160, 1994.

[Bou94] Boulton, R. J.: A pretty-printer specification language with support for repetitive nested structures. In *Proceedings of the International Conference on Technical Informatics (ConTI'94)*, volume 5, pages 226–235, Timişoara, Romania, November 1994.

[BoS94] Bornat, R. and Sufrin, B.: Jape: a literal, lightweight, interactive proof assistant. Technical Report 641, Queen Mary and Westfield College, University of London, 1994.

[BSH93] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A.: Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[BeT98] Bertot, Y. and Théry, L.: A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25:161–194, 1998.

[BVW97] Backhouse, R. C., Verhoeven, R. and Weber, O.: Mathspad: a system for on-line preparation of mathematical documents. *Software – Concepts and Tools*, 18:80–89, 1997.

[BuW81] Bundy, A. and Welham, B.: Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation. *Artificial Intelligence*, 16:189–212, 1981.

[CAB86] Constable, R., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harber, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T. and Smith, S. F.: *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[CaH95] Caferra, R. and Herment, M.: A generic graphic framework for combining inference tools and editing proofs and formulae. *Journal of Symbolic Computation*, 19:217–243, 1995.

[Cle90] Clément, D.: A distributed architecture for programming environments. *Software Engineering Notes*, 15(5), 1990. *Proceedings of the 4th Symposium on Software Development Environments*.

[DFH93] Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C. et al.: *The Coq Proof Assistant User's Guide*. INRIA, May 1993. Version 5.8.

[DéR94] Déry, A.-M. and Rideau, L.: Distributed programming environments: an example of message protocol. Rapport Technique 165, INRIA, 1994.

[GRB93] Goldson, D., Reeves, S. and Bornat, R.: A review of several programs for the teaching of logic. *The Computer Journal*, 36(4):374–386, 1993.

[JMB93] Jacobs, I., Montagnac, F., Bertot, J., Clément, D. and Prunet, V.: The Sophtalk reference manual. Rapport Technique 150, INRIA, 1993.

[LBM98] Lowe, H., Bundy, A. and McLean, D.: The use of proof planning for cooperative theorem proving. *Journal of Symbolic Computation*, 25:239–261, 1998.

[MCC86] Morcos-Chounet, E. and Conchon, A.: Ppml: A general formalism to specify pretty-printing. In H.-J. Kugler, editor, *Information Processing 86 (Proceedings of IFIP Congress)*. IFIP, Elsevier Science Publishers B.V. (North-Holland), 1986.

[MeH97] Merriam, N. A. and Harrison, M. D.: What is wrong with guis for theorem provers? In *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, September 1997.

[MaN94] Magnusson, L. and Nordström, B.: The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes Computer Science*, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[MTH90] Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*. MIT Press, 1990. ISBN 0-262-63132-6.

[OEC97] Ozols, M. A., Eastaughffe, K. A. and Cant, A.: Xisabelle: A system description. In *Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 400–403. Springer-Verlag, July 1997.

[PBR98] Pons, O., Bertot, Y. and Rideau, L.: Notions of dependency in proof assistants. In *User Interfaces for Theorem Provers 1998*, Eindhoven University of Technology, 1998.

[Pon97] Pons, O.: Undoing and managing a proof. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1997"*, Sophia-Antipolis, France, 1997.

[Rit88] Ritchie, B.: *The design and implementation of an interactive proof editor*. PhD thesis, University of Edinburgh, 1988.

[ReV97] Rémy, D. and Vouillon, J.: Objective ML: A simple object-oriented extension of ml. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pages 40–53, Paris, France, January 1997.

[Sup84] Suppes, P.: The next generation of interactive theorem provers. In *Automated Deduction (CADE-7)*, volume 170 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1984.

[Sym95] Syme, D.: A new interface for hol - ideas, issues, and implementation. In *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 324–339. Springer-Verlag, September 1995.

[TBK92] Théry, L., Bertot, Y. and Kahn, G.: Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5), 1992. Proceedings of the 5th Symposium on Software Development Environments.

[Thé98] Théry, L.: A certified version of Buchberger's algorithm. In *Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, July 1998.

[Thi90] Thimbleby, H.: *User Interface Design*. Frontier Series. ACM Press, 1990.

Received October 1998
Accepted in revised form May 1999