Generating Graphical User Interfaces in a Functional Setting

Kolyang, C. Lüth, T. Meyer, B. Wolff

Bremen Institute for Safe Systems, TZI, FB 3
Universität Bremen
Postfach 330440
28334 Bremen
Germany
{kol,cxl,tm,bu}@informatik.uni-bremen.de
Phone: +49 (421) 218 7585, Fax: +49 (421) 218 3054

June 17, 1996

Abstract

We present a new approach to implementing graphical user interfaces (GUIs) for theorem provers and applications using theorem provers. A typed interface to Standard ML from Tcl/Tk provides the foundations upon which a generic user interface is built. Besides the advantage of type safeness, this technique yields access to the full power of the modularization concepts of Standard ML. It leads to a generic GUI, which instantiated with a particular application yields a GUI for this application. We present a prototypical implementation with two instantiations: an interface to Isabelle itself and a system for transformational program development based on Isabelle.

1 Introduction

Graphical user interfaces have been identified as a major potential to increase the usability and productivity of interactive theorem provers (like HOL [GM93] and Isabelle [Pau94]) and formal program development tools [HK93, Smi91]. The question of how to hide the theorem prover's internals in an easy-to-use interface which is accessible to the user with little experience in formal logic becomes essential for their wider use in research and industry.

The context of this work is the UniForM project [KPO+95] (funded by the German Ministry for Education and Research), the aim of which is to develop a *Universal Formal Methods Workbench*, a framework integrating different formal methods into one workbench that is useable in practice. Its logical basis is higher-order logic, into which different formal methods such as CSP [Hoa85] and Z [Spi92] (and others) are encoded. Emphasis is put on transformational program development. The theorem prover Isabelle is used to prove correctness of a particular development as well as correctness of the transformation rules (see [KSW96]). Since one of its main objectives is to enable non-expert users to actually perform at least part of the development themselves, there is a crucial need for graphical user interfaces and advanced techniques of their implementation.

In this paper, we present a prototypical implementation of a graphical user interface to Isabelle. Since Isabelle is used in more than one way (proving transformational program developments correct, proving transformations correct, etc), we require a generic interface which can be instantiated with different applications based on Isabelle. The system is entirely implemented in Standard ML, in order to use of SML's powerful module system and moreover achieve a type-safe interaction with Isabelle. The user interface is implemented using the Tk toolkit, for which we have developed an encapsulation into SML called sml_tk.

We will first examine the system architecture, followed by the look and feel of the system. The final section contains comparisons with related work and the obligatory list of future work.

2 System Architecture

We will below briefly touch on all components of the system architecture (Figure 1) in turn: at first highlighting why Isabelle's system architecture makes it so amenable to extensions such as the present, then examining the functional encapsulation sml_tk of Tcl/Tk, and finally describing the generic GUI, with two possible instantations— the graphical user interface to Isabelle itself, and the Transformation Application System TAS.

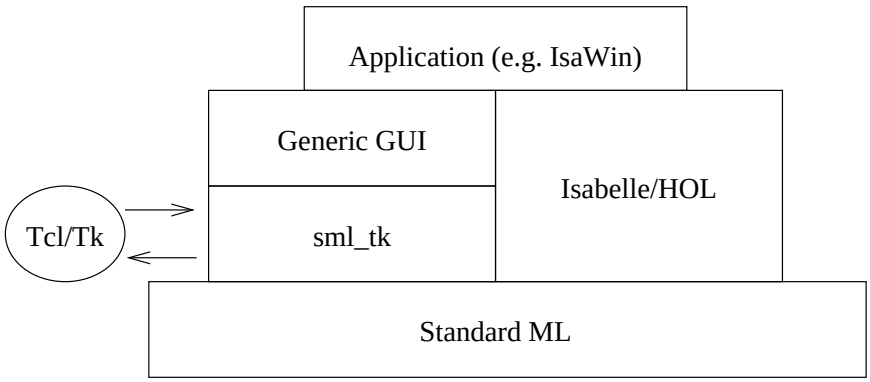

Figure 1: System Architecture

The System Architecture of Isabelle

Isabelle essentially consists of a collection of ML types for objects such as theorems, proofs, rewriting sets, theories and tactics, and ML functions to apply tactics (to prove a goal), start and finish proofs, manipulate and handle the proof state, and so on, organized into a collection of ML structures and functors. In the LCF tradition, ML is used as the command language and user interface: the user types a term, which is evaluated by the ML run time system.

One can extend Isabelle conservatively by writing ML functions, using the abstract datatypes provided by Isabelle. ML's typing discipline and structuring mechanisms protect the theorem-proving core of Isabelle from being logically corrupted, and provide a closely coupled and safe (in particular, typed) interaction with Isabelle. We consider this to be a great advantage of LCF style theorem provers.

sml_tk

The user interface description and command language Tcl/Tk [Oos94] has seen a tremendous success in the recent years. It offers a highly portable interface to window systems such as the X Window System, to which it is mainly geared. It consists of the toolkit Tk and the command language Tcl, a Lisp-like scripting language with only one datatype strings. The Tk toolkit offers in our opinion the right level of abstraction to access a window system, but the programming language Tcl is not designed to write large applications in, but rather facilitate communication between them. Hence to use the Tk toolkit for an application written in ML, there is a need for an interface from ML to Tcl/Tk.

The sml_tk package is a portable, typed encapsulation of Tcl/Tk into Standard ML. It provides abstract ML datatypes for the Tcl/Tk objects. For example, one of the main Tcl/Tk objects is a *widget*, which is the basic all-purpose building block for graphical objects. A widget can be a simple button, a text display, a so-called frame which contains other widgets, and many more. Its appearance can be modified using *configuration options* defining the font used, the text displayed, size, possible outlines drawn around it, etc. The corresponding ML datatype reads (abbreviated)

```
datatype Widget = Button of WidId * Configure list * ...
                | Label of WidId * Configure list * ...
```

```
                | Frame of WidId * Widget list * ...

datatype Configure = Text of string | Command of (unit-> unit)
                   | Width of int   | ...
```

(where `WidId` is a string uniquely identifying the widget). A very simple widget consisting of a a short text and a button, which when pressed triggers a function `closeWin` (of type `unit->unit`) would be described by the following ML expression:

```
Frame("fr1", [Label("lab0", [Text "A Simple Widget"]),
      Button("butt0", [Command closeWin])])
```

The sml_tk package translates ML expressions such as the above into Tcl code that is sent (via a pipe) to the Tcl interpreter where it is executed. It further offers a collection of structures providing standard components of a user interface such as error and warning windows, input windows and a file browser.

Another encapsulation of Tcl/Tk in ML (albeit for a slightly different flavour of ML) is the Caml/Tk package [PR95]. There are two main differences of Caml/Tk to sml_tk: in Caml/Tk, the Tk datatypes such as widgets are constructed by repeatedly applying functions with side-effects, whereas in sml_tk, Tk datatypes are freely generated ML datatypes, resulting in clearer and more readable code. Further, Caml/Tk interfaces the Tk library directly, by linking it into the executable. This has the advantage of being faster, but the disadvantage of being highly dependent on the interface of the Tk library at a very low level of abstraction, and processes blocked in the Tk library can inadvertently and unnecessarily block the calling ML process as well.

The Generic Graphical User Interface

The Generic GUI uses the interface description facilities provided by sml_tk to provide a generic graphical user interface. Its main components are a module allowing the user to manipulate *items* (graphical objects) on a *canvas* (a window area to draw on) by "grabbing" them with a cursor, moving them across the screen or "dropping" them onto other objects (thereby possibly triggering an operation), and a module giving a semantics to these items with respect to a given application as described below.

An application can be abstractly characterized as follows:

- It has *objects*, each of which has a *type*. The type determines which operations are applicable to this object, and is indicated by the object's *icon*.
- For each of the objects, the application provides a dictionary of unary operations, comprising *standard operations* (display an object, delete an object etc.) and operations specific to the object type.
- There is a dictionary of binary operations, corresponding to the operations triggered by dragging and dropping objects (indexed by the types of the dragged and dropped object) the result of which is a list of objects produced by the operation. Further, there is a nullary initialization function, which returns the list of objects to initially appear on the workspace.

Hence, applications can be described by an ML signature:

```
signature APPL_SIG =
  sig
    type objtype
    type object

    val objType : object -> objtype            (* typing function *)

    val init    : unit   -> object list        (* initial objects *)
```

```
    val delete  : object -> unit                (* unary operations *)

    val monOps  : objtype -> (object-> unit) list (* further unary opns. *)

    val binOps  : objtype * objtype -> (object-> object-> object list)
                                                (* binary operations *)
  end
```

The real signature of course is far more elaborate, containing in particular details about the visual appearance (such as the size of the window, the particulars of the icons depicting the objects and their locations etc). The Generic GUI is implemented as a functor

```
functor GenGUI(structure appl: APPL_SIG ) = ...
```

which, when instantiated with an application, returns a graphical user interface for that application.

We will now look at two example applications. The first is the Isabelle graphical user interface IsaWin, the second the Transformation Application System TAS.

IsaWin as an Instantiation of the Generic GUI

For IsaWin, the object types are

- theorems (corresponding to Isabelle's theorem type),
- proofs (corresponding to Isabelle's proof states),
- two types of rewriting sets (called simplifier sets and classical simplifier sets; the latter can only be used with classical logics),
- and theories (collections of type declarations, theorems and rewriting sets).

The operations include

- backward resolution by dropping a theorem onto a proof,
- forward resolution by dropping a theorem onto a theorem, or
- rewriting by dropping a rewrite set onto a proof.

There is more than one possibility for backward resolution and rewriting, leading to a refinement of the drag&drop paradigm by introducing the concept of the construction area (see below).

There is no possiblity to graphically edit tactics yet. In Isabelle, more advanced tactics can be constructed from the basic ones (such as backward resolution and rewriting) by using *tacticals* (*e.g.* `REPEAT`, which means "do until failure"), and one could think of graphical support for this process. We have not done this, and at the present are not considering it, because our system is aimed more at the beginning user.

TAS as an Instantiation of the Generic GUI

The Transformation Application System TAS (formerly YATS [KSW96]) allows the transformational development of programs. It is based on Isabelle, combining the flexibility of Isabelle with the program development principles of conventional program transformation systems like PROSPECTRA [HK93] or KIDS [Smi91]. Briefly, it contains *transformation rules*, each of which is essentially a tactical program (called the *tactical sugar*) controlling the application of a *logical core theorem* of the form

$$\forall P_1, \ldots P_n . A \Rightarrow I \rightsquigarrow O$$

where $P_1, \ldots, P_n$ are the *parameters* of the rule, A the *applicability condition*, I the *input pattern* and O the *output pattern*. $\rightsquigarrow$ is the *transformation relation*, which is an arbitrary transitive and reflexive relation. The transformation is *applied* by performing a backward resolution with

the transitivity of $\rightsquigarrow$, and executing the tactical sugar. This leads to unification of the present goal (corresponding for example to a specification or program) with the input pattern I, and, if both unification and application of the tactic succeed, results in a proof state containing the substituted output pattern O, and the substituted applicability condition A. This corresponds to the transformed specification or program, along with the proof obligation of the applicability of the transformation rule.

The Transformation Application System is designed to keep everything about proofs in Isabelle away from the user. The proof obligations resulting from the application condition A are proven using another interface to Isabelle (like IsaWin), such that the user does not have to worry about the details of how the transformational process is implemented within Isabelle, leaving her with the main design decisions of transformational program developement: which rule to apply, and how to instantiate its parameters.

With typical transformation rules, parameter instantiations are lengthy enough to merit a dedicated object type (to avoid having to retype them, allow copying them etc.) Hence, the object types are transformational program developments (corresponding to Isabelle proof states), transformation rules and parameter instantiations.

The operations include applying a transformation rule by dropping it onto a transformational program development, and instantiating the parameters of a transformation rule by dropping the instantiation on the rule.

3 Visual Appearance

In general, the main window is divided into two parts, the *assembling area* in the upper part, and the *construction area* in the lower part. In the assembling area, one can manipulate objects without regards to their internal state. In the construction area, one can manipulate an object using, and moreover altering, its internal state. In the IsaWin instance, for example, we can abstractly manipulate theorems by performing forward resolution between them (obtaining a new theorem and leaving the two arguments untouched), but to manipulate a proof we have to know about its internal state like its subgoals and its main goal, and every application of a tactic will alter the internal state.

Each application is geared towards one particular object type, called the *construction objects*. These are the only ones that can be "opened" and manipulated on the construction area. For IsaWin, these are proofs, and for TAS, these are transformational program developments. The idea is that these are the objects "constructed" by the application.

We will now describe the visual appearance of the Generic GUI instantiated with IsaWin as shown in figure 2.

The Assembling Area

The assembling area mainly consists of objects, represented by icons, where different kinds of icons correspond to different object types. They can be moved and dropped, in particular onto each other and into the construction area to trigger an operation with the object under construction (for example, dropping a theorem onto the construction area will cause a backward resolution step).

For each object there is a popup menu activated by the right mouse button, containing entries for unary operations like deleting and showing the object, and possibly entries for object type specific operations. For instance, the popup menu of an Isabelle theory object contains an entry to start a proof in that theory.

The assembling area further has a menu bar with entries for importing and exporting objects. Here, a browser allows to search for particular theories and theorems by name via regular expressions. All items found are displayed in a separate window, from where they can be dragged onto the assembling area to work with them. This menu bar is specific to the application.

The Construction Area

By double-clicking, construction objects are opend and appear in the construction area. Every application comes with its own construction area. The main part of the IsaWin construction

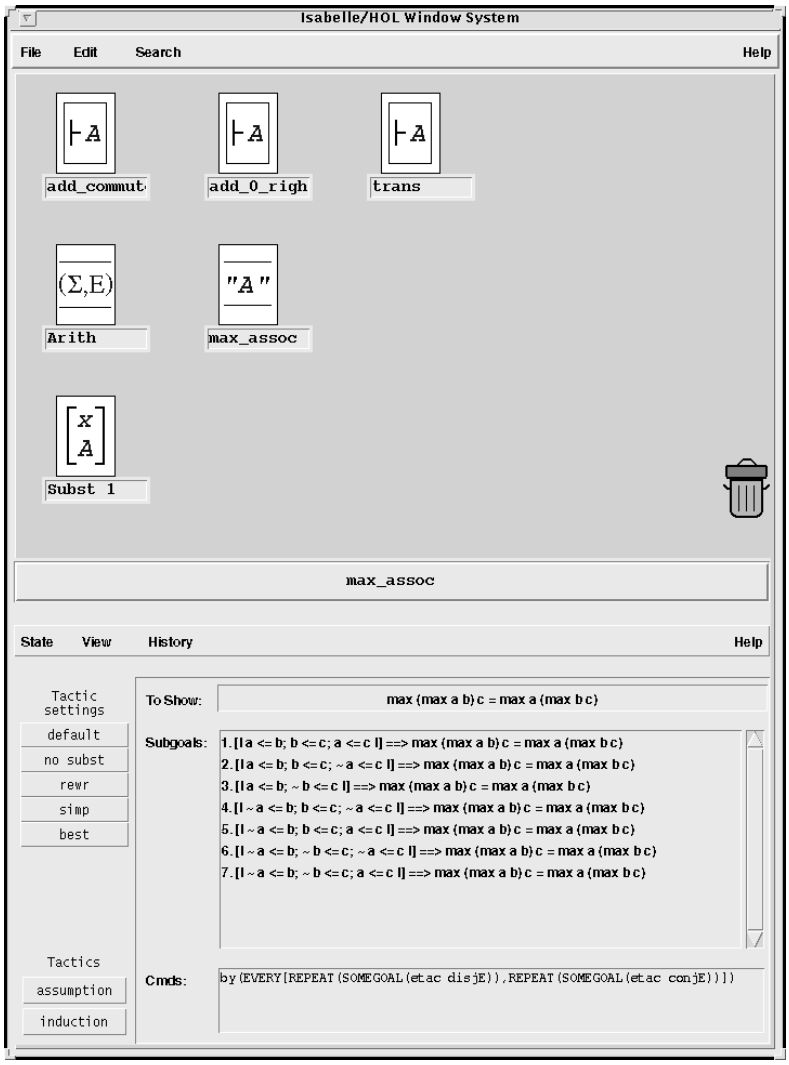

Figure 2: The Main Window

area shows the initial goal that has to be proven, the current subgoals (if there is not enough space to display all subgoals a scrollbar is provided to scroll through them) and the last ML command executed by Isabelle. In that area, the user can also type ML commands to be executed by Isabelle. The display of the initial goal and the last ML command can be toggled (via the view menu) to allow more space for the subgoals.

On the left-hand side there are various menues to determine the tactic settings. For example, Isabelle offers three different kinds of backward resolution: normal resolution, elim-resolution and destruct-resolution, each corresponding to a specific tactic. With the first button one can select which tactic to apply whenever a theorem is dragged down from dropped in the construction area.

The other settings determine whether explicit substitutions should be used, and similar selections between different tactics to apply rewriting rules, simplifier sets and classical simplifier sets.

The menu bar of the construction area contains the state menu, the view menu and the history menu. With the state menu contains, one can end a proof, introducing the proven goal on the assembling area as an Isabelle theorem or close the construction area (this can also be done while the proof is still incomplete), and with the history menu one can go back and forth one step in the proof, or display its complete history.

4 Related Work, Outlook and Conclusion

Related Work

Other systems like TkHOL [Sym95] and XIsabelle [CO95] are clearly superior to ours as far as functionality and user-friendliness is concerned. However, they are predominantly implemented in Tcl and not in ML. We believe that our approach offers a greater flexibility, leading to a higher extendability and reusability.

Future Work

This work is only in its early stages, and in this section we will sketch a few of the improvements we have in mind for our system.

- Error handling:

  Probably the most important area in which future work needs to be done is error handling. Roughly, there are three kinds of errors we can distinguish:

  - Critical errors, such as Isabelle (or the ML interpreter) being unable to parse a theory file or command line.
  - Failures, such as when the application of a tactic or unification fails.
  - Lack of success ("Why doesn't this work?"), such as when a tactic or resolution can be applied but the outcome is not what the user expects, or when the simplifier fails to solve/simplify/rewrite as the user would like it to.

  Errors of the first kind have been nearly eliminated by restricting the access to Isabelle. For the second and third kind, satisfactory error handling is most needed, showing where the unification failed, or precisely at which point the tactic could not be applied. The definition of these "points of failure" may not be exact but requires some heuristics; the general philosophy here should be to give a good guess of the reason with a brief explanation, from where the user can interactively obtain more information (rather than just printing any information that can possibly be useful).

- Pretty Printing:

  We envisage a pretty printer which converts the symbols of Isabelles logics into graphics which are easier to read; for example, instead of

  ```
  [| !! x. P(x); P(x) ==> R |] ==> R
  ```

  we would like the user to see and edit

  $$\frac{\forall x.P(x) \quad P(x) \Rightarrow R}{R}$$

  A tool to display and edit structured formulae in such a way is currently being developed at Universität Bremen.

- User Guidance:

  One of the nice features of the XIsabelle interface is that it displays all tactics which one can currently apply to a subgoal. A feature in this vein is surely desirable, extended to simplifier sets and rewrite rules.

- Proof Editing:

  Proof scripts are provided in an early stage: commands sent to Isabelle are recorded. The next step is to translate these scripts, allowing the replaying of the whole development. A further step would be to parse the proof scripts and treat proofs as Isabelle objects with an associated theory of their algebraic properties (for example, stating when two transformation rules can be interchanged).

Concluding Remarks

We have shown how to implement a graphical user interface for Isabelle with a generic system architecture. We have exploited the advantages of the Tcl/Tk toolkit, but avoided its disadvantages by encapsulating it into ML, and made use of the structuring powers of the ML module system. The result is a prototypical implementation of a user interface which can be extended and customized fairly easily.

The generic nature of the system allowed by the expressiveness of the ML module system is useful because we can build a graphical user interface to any application based on Isabelle, like TAS, by merely instantiating a functor. By allowing the development of easier-to-use prover applications, we hope this will help to open up and develop new fields for theorem proving in the future.

References

[CO95] A. Cant and M. A. Ozohls. XIsabelle: A graphical user interface to the Isabelle theorem prover. Technical report, Defence Science and Technology Organisation, Salisbury, Australia, October 1995. `ftp://ftp.cl.cam.ac.uk/ml/XIsabelle.2.0.tar.gz`

[GM93] M. J. C. Gordon and T. M. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logics.* Cambridge University Press, 1993.

[HK93] B. Hoffmann and B. Krieg-Brückner. *Program Development by Specification and Transformation.* Number 690 in Lecture Notes in Computer Science. Springer Verlag, 1993.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[KPO+95] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, and A. Baer. UniForM Workbench — Universelle Entwicklungsumgebung für formale Methoden. Technischer Bericht 8/95, Universität Bremen, 1995. English version: UniForM Workbench — Universal Formal Methods Workbench. `http://www.informatik.uni-bremen.de/~uniform/`

[KSW96] Kolyang, T. Santen, and B. Wolff. Correct and user-friendly implementations of transformation systems. In M. C. Gaudel and J. Woodcock, editors, *Formal Methods Europe*, number 1051 in Lecture Notes in Computer Science, pages 629–648, Oxford, 1996. Springer Verlag.

[Oos94] J. K. Oosterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Pau94] L. C. Paulson. *Isabelle - A Generic Theorem Prover.* Number 828 in Lecture Notes in Computer Science. Springer Verlag, 1994.

[PR95] F. Pessaux and F. Rouaix. The Caml/Tk interface. Technical Report 971, INRIA Rocquencourt, May 1995. `http://pauillac.inria.fr/camltk/camltk.html`

[Smi91] D. R. Smith. KIDS — a semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024– 1043, 1991.

[Spi92] M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992. 2nd edition.

[Sym95] D. Syme. A new interface for HOL— ideas, issues and implementation. In *Higher Order Logic: Theorem Proving and its Applications*, number 971 in Lecture Notes in Computer Science, pages 325– 339. Springer Verlag, 1995.