Generic Interfaces for Formal Development Support Tools

Kolyang, C. Lüth, T. Meyer, B. Wolff

Bremen Institute for Safe Systems, TZI, FB 3
Universität Bremen
Postfach 330440, 28334 Bremen
{kol,cxl,tm,bu}@informatik.uni-bremen.de

Abstract. We present a new approach to implement graphical user interfaces (GUIs) for theorem provers and formal development support tools. A typed interface between Standard ML and Tcl/Tk provides the foundations, upon which a generic GUI is built. Besides the advantage of type safeness, this technique yields access to the full power of the modularisation concepts of Standard ML: the generic GUI is a functor (a parametric module), 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

In this paper, we present a new approach to implement graphical user interfaces (GUIs) for formal program development systems like transformation systems or interactive theorem provers. Its distinguishing feature is a generic, open system design which allows the development of a family of tools for different formal methods on a sound logical basis with a uniform appearance. This is particularly important since we share the view that "there is no single theory for all stages of the development of software (...)" [Hoa96].

The context of this work is the UniForM project [KPO+95], the aim of which is to develop a framework integrating different formal methods and formal methods tools. We distinguish two major classes of tools to be integrated:

- *Direct Tools:* By this class we mean tools that directly implement the rules of a logic or a particular proof method. Such tools tend to be difficult to modify and to formally reason about, but can possess remarkable proof power in specific problem domains (e.g. *FDR* [For95]) and comfortable user interfaces. For these reasons, they seem to meet a greater acceptance in industry.
- *Logical Embeddings:* Formal methods such as CSP and Z can be logically embedded into an LCF-style tactical theorem prover such as *HOL* [GM93] or *Isabelle* [Pau94]. Coming with an open system design going back to Milner, these provers allow for user-programmed extensions in a logically sound way. Their flexibility, generality and expressiveness makes them *symbolic programming environments* that are hardly considered as a tool at all in industry.

Since Logical Embeddings play an important role in UniForM and since it is one of the project objectives to enable non-expert users to actually perform at least part of the developments themselves, there is a crucial need for an encapsulation technique of tactical theorem provers to specific tools including graphical user interfaces and advanced techniques of their implementation.

In this paper, we show a general approach to developing generic encapsulations. Given a Logical Embedding of a formal method into the LCF prover Isabelle, we provide a generic technique to generate a graphical user interface. We demonstrate the viability of our approach with two case studies. One is an implementation of a graphical user interface for Isabelle itself, offering a deliberately limited abstraction to the underlying Isabelle environment. The other is the transformation system TAS following the prominent approaches of CIP [BBB+85], PROSPECTRA [HK93], and, in particular, KIDS [Smi91].

The Transformation Application System TAS offers a high abstraction to Isabelle, a simple design and proven correct transformations which are easy to extend and to modify. As object language for TAS, we chose higher order logic (HOL) which is one instantiation of the generic system Isabelle with an object logic and is delivered with the standard package. A subset of HOL formulas can easily be translated into functional programs (e.g. HASKELL or ML).

The visual result of an encapsulation is a graphical user interface. Aiming at support for a multitude of formal methods, a generic approach can also provide some unity in the conceptual modelling, the visual presentation, the accessibility of functionality and the interchangeability within the resulting tool family. With respect to the underlying programming techniques, this leads to the requirement of an open system design, structured in small components with high reusability and customizability, integrated in a language with higher abstraction and modularization concepts. Hence, it is not accidentally that our work integrates three existing and well documented public domain systems: the tactical theorem prover Isabelle, based on Standard ML [Pau91] and the X-Window toolkit Tk [Ous94].

As a result of our work, we claim that our implementation technology is applicable to a fairly wide range of formal development support tools, including Direct Tools and combinations of Logical Embeddings with them. We explicitly encourage other researchers and research groups to copy, modify and extend our prototype implementations according to their needs.

This paper proceeds as follows: we first give a brief description of the necessary concepts and techniques to perform transformational program development inside Isabelle. We will outline the generic system architecture, and show how it can be instantiated to a system TAS for transformational program development and a graphical user interface IsaWin for a theorem prover. The final section contains comparisons with related work and an outline of future work.

2 Transformational Program Development in Isabelle

Modelling transformational program development is by no means a new idea. In this section, we will describe our approach, following the lines of [KSW96a]. A transfor-

mational development is described by the sequence of specifications SP_i

$$SP_1 \rightsquigarrow \ldots \rightsquigarrow SP_n$$

In a full transformational development SP_1 is the requirement specification of a program and SP_n the executable specification (from which a program can be generated), but a transformational development may describe any subsequence of this as a single step in the overall development process, deriving a more refined specification or program from a more general one.

Thus one can abstractly view the SP_i as arbitrary formulae, and a development step $SP_i \rightsquigarrow SP_{i+1}$ is given by the application of preconceived rules called *transformations*, ranging from simple logical rules to complex ones that convert a certain design pattern into an algorithmic scheme (such as *Global Search* or *Divide & Conquer*; see [KSW96a] for more details).

The basic idea of the Transformation Application System TAS is to separate the *logical core* of a transformation from the pragmatics of its application, its *tactical sugar*. By proving the logical core theorem in the logics of the object language, a transformation can be proven correct. The tactical sugar, often highly system dependent, drives the concrete application in a development context (see below). The distinction between logical core theorem and tactical sugar establishes an important separation of concerns.

A logical core theorem generally has the following form:

$$\forall P_1, \ldots, P_n \,.\, A \Rightarrow I \rightsquigarrow O \tag{1}$$

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 transitive and reflexive *transformation relation*.

In Isabelle, tactical sugaring can be based on the concept of *meta variables* that can be seen as *holes* in a formula to be filled in later via substitutions. In practice, most of these substitutions are constructed automatically via higher order unification during resolution steps (as in PROLOG).

A transformational development is started by creating an initial proof state

$$1.\ SP_1 \rightsquigarrow ?Z$$

where $?Z$ is the Isabelle notation for a meta variable representing the final result.

A transformation given by the core theorem 1 is *applied* by performing the following sequence of tactical operations: first, a resolution with the transitivity of $\rightsquigarrow$ is carried out. This leads to a proof state with two logically conjoined so-called *subgoals*:

$$1.\ SP_1 \rightsquigarrow ?Y$$
$$2.\ ?Y \rightsquigarrow ?Z$$

where $?Y$ is the new intermediate specification and $?Z$ remains the ultimate target of the development. In the second step, $?Y$ is substituted by the transformed specification SP_1: by forward chaining[1] of the logical core theorem 1 and the elimination

[1] Forward chaining is the composition of rules: from $\frac{A}{B}$ and $\frac{B}{C}$, we derive the rule $\frac{A}{C}$.

rules for the universal quantifier (allE) and the implication (mp)

$$\frac{\forall x.P(x)}{P(?a)}\ \mathsf{allE} \qquad \frac{?A \Rightarrow ?B \quad ?A}{?B}\ \mathsf{mp}$$

one obtains the logical core theorem in a form where the variables bound by the universal quantifier are substituted for meta variables. We now unify the input pattern I of the transformation rule with the specification SP_1 (by resolving the transformed core theorem with subgoal 1), and obtain a substitution σ such that $\sigma(I) = SP_1$. Applying this substitution to the output pattern yields the transformed program $SP_2 \stackrel{def}{=} \sigma(O)$, and applying it to the applicability conditions yields the proof obligation $A' \stackrel{def}{=} \sigma(A)$. The proof obligations will appear as a new subgoal, since they also need to be proven to make the transformation sound. The proof state after applying the transformation thus reads

1. A'
2. $SP_2 \rightsquigarrow ?Z$

The tactical sugar as described above can vary in many respects. For example, it might contain standard proof procedures which remove trivial proof obligations, or it may use more sophisticated, semantic matching techniques [Shi96].

The application of the next transformation will be focused on subgoal 2 and so forth. This way, transformational developments can be represented within the infrastructure of Isabelle, allowing browsing and copying developments and abstract operations on them.

A development is closed by resolving with the reflexivity law $?A \rightsquigarrow ?A$ which simply unifies the current result SP_i of the transformational development with the final result $?Z$. Note that this does not imply that the specification is executable in any sense, it just allows termination of the transformational development.

The Transformation Application System is designed to hide these internal tactical steps, the existence of meta variables etc. from the user. Moreover, the proof obligations may be simplified by refined, transformation-specific tactical sugar or given to external decision procedures or other interfaces to Isabelle (like IsaWin) which are designed to perform technical, logic-oriented proofs.

The user of the Transformation Application System will not have to worry about the details of how the transformational process is implemented within Isabelle, and the proof of side conditions can be deferred to a later stage. This allows the user to concentrate on the main design decisions of transformational program development: which transformation to apply, and how to instantiate its parameters.

3 Generic System Architecture

We will below briefly touch on all components of the system architecture (Fig. 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 instantiations of the application— the graphical user interface IsaWin to Isabelle itself, and the Transformation Application System TAS.

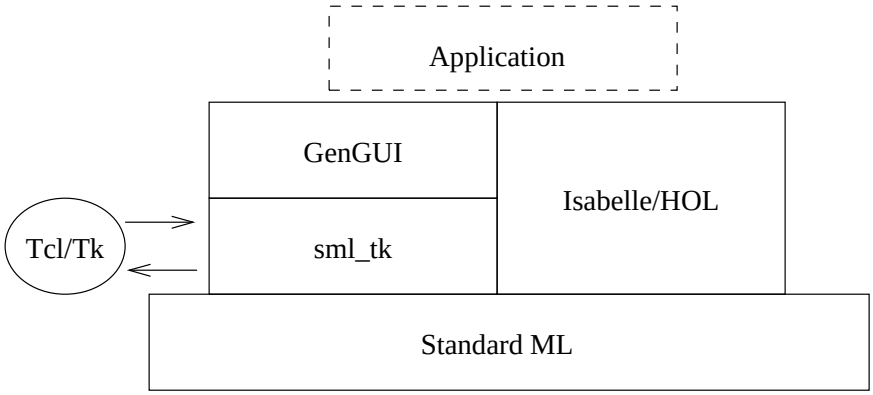

Fig. 1. System Architecture

3.1 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, organised 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.

3.2 sml_tk

The user interface description and command language Tcl/Tk [Ous94] 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 scripting language with strings as the only datatypes. 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. [2] 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, developed at the University of Bremen [LWW96]. It provides abstract ML datatypes for the Tcl/Tk objects. For example, one of the main Tcl/Tk objects is a

[2] Recently, extensions to Tcl such as [incr-Tcl] have appeared, but they offer nothing close in terms of abstraction and structuring concepts than provided by typed, functional languages such as SML or Haskell.

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 visual appearance can be modified using *configuration options* defining the font used, the text displayed, size, possible outlines drawn around it, etc. The corresponding sml_tk 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 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.

Detailed information on sml_tk can be found in [LWW96], or at the sml_tk home page (`http://www.informatik.uni-bremen.de/~cxl/sml_tk/`).

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 the calling ML process can inadvertently and unnecessarily be blocked in the Tk library.

3.3 The Generic Graphical User Interface GenGUI

The module GenGUI 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.

An application can be abstractly characterised 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 object type, 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 initialisation function, which returns the list of objects to initially appear on the workspace.
– Each application is geared towards one particular object type, called the *construction objects*. These can be "opened" and manipulated in a particular area of the window, the *construction area*. The idea is that these are the objects "constructed" by the application. This manipulation will often take further user interaction, so the whole construction area is provided by the application as one single *widget* (see Sect. 4 below).

The difference between the unary and binary operations applied by drag& drop and the operations available via the construction area is that the first only allow an abstract manipulation of objects, without regard to their internal structure, and the second allow manipulation using, and moreover altering the internal structure. These restrictions are enforced by the typing of the operations – a function from the dictionary of binary operations cannot change the arguments it is passed. The function which opens the construction area, on the other hand, gets passed a continuation (of type `object-> unit`) as its second argument. After the manipulation of the object in the construction area has finished, the continuation is called with the changed object, and hence its state can change.

Consider a very simple example in which the objects are texts. Then dropping a text onto another one would produce a new text object, which is the dropped text concatenated to the one it has been dropped onto. Both argument texts would remain unchanged. The construction area would be a very simple text editor, allowing to change the text.

Applications are 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          (* std. unary opn. *)


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

    val binOps  : objtype * objtype ->
```

```
                                    (object-> object-> object list)
                                            (* binary operations *)

    val openWS  : object -> (object-> unit)-> TkTypes.Widget
                                   (* open the Construction Area *)
  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).

GenGUI itself is a functor

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

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

3.4 TAS as an Instantiation of the Generic GUI

The construction objects will be transformational program developments, corresponding to Isabelle's proof state, tagged with a protocol of the (abstract) tactical operations that lead to the proof state. This design decision leads already to a concrete implementation of undo- and replay operations. Since in realistic transformation rules (such as Global Search presented in [KSW96a]) parameter instantiations are lengthy, instantiations (i.e. substitutions) merit a dedicated object type to avoid retyping and allowing copying them etc. Hence, the object types are

- transformational program developments,
- transformation rules with their parameters not instantiated,
- transformation rules with their parameters instantiated,
- and parameter instantiations.

The binary operations are

- applying a transformation rule. This comes in two varieties, one for instantiated and one for uninstantiated rules. In the latter case, the user is prompted for an instantiation of the parameters;
- instantiating a transformation rule by dropping an instantiation on a transformation rule.

The construction area is rather simple. It will show the state of the object currently under construction: the starting specification or program, the current program or specification, and the current proof obligations. One applies a transformation by dragging a transformation rule into the construction area.

3.5 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.

The construction objects here are proofs. The construction area is rather elaborated, since the backward resolution and rewriting require some user control. Isabelle offers three kind of backward resolution, and two kinds of rewriting tactics between which the user can choose. Further, the tactics can selectively be applied to one subgoal, and the user can browse the proof history forwards and backwards.

IsaWin offers also a more abstract access to Isabelle's operations. An example is the induction operation that attempts to choose an appropriate induction scheme from a database and performs induction on a variable of an inductive type (such as natural numbers). Other examples of these abstract interface operations could be fold/unfold operations or a case split operation.

3.6 Implementation Issues

Table 1 gives an impression of the general size of the code. The sml_tk code is the largest chunk, mainly because sml_tk is slowly approaching the full functionality of Tcl/Tk. Building on that, TAS and IsaWin are fairly compact.

Module	Code size (lines of SML)
sml_tk	7580
GenGUI	1300
IsaWin	3000
TAS[a]	2570

[a] TAS and IsaWin share about 1000 lines of code.

Table 1. Size of Code

The code as produced by the Standard ML of New Jersey compiler offers satisfactory response times, but shows a voracious appetite for memory. A running system will need at least 32 Megabytes of memory, and to compile TAS and IsaWin 64 Megabytes or more are required (on a Sun SPARC or UltraSPARC workstation). This is mostly due to Isabelle; sml_tk itself compiles on far smaller machines.

4 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. The assembling area contains the icons representing the objects. One can select unary operations via a *pop-up menu* activated by the right mouse button, or drop one object onto another, selecting a binary operation.

Further, every instantiation has a *trash can*: a special object, supplied by GenGUI, which deletes any object dropped into it.

Figure 2 shows the main window of the instantiation of GenGUI with the Transformation Application System. The proof obligations do not all fit into the construction area, so a scrollbar is provided to scroll through them. The construction area further has a close button, which "closes" the object currently under construction, so it reappears as an icon in the assembling area. This allows the user to work on several program developments concurrently.

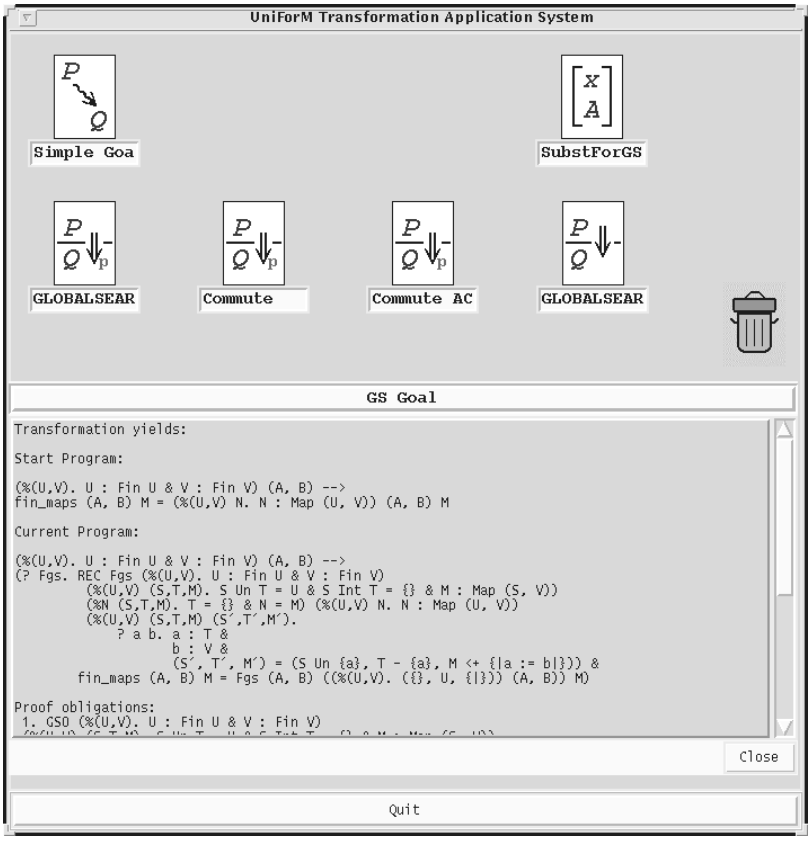

Fig. 2. TAS Main Window

In contrast, the construction area of the IsaWin instantiation shown in Fig. 3

has more bells and whistles. Briefly, the buttons on the upper left side of the construction area are the ones determining tactic settings (*e.g.* which of the different rewriting tactics to apply), and the buttons on the lower left side supply special tactics. The main part of the construction area shows the main goal at the top, the subgoals currently to be proven in the middle, and the last Isabelle command executed beneath. Menus offer further operations, such as the proof history browser, a theorem browser and help functions. In Fig. 3, the history browser is currently open in a separate window in the upper right hand of the screen, and the theorem browser is currently showing in a separate window on the left side.

There are two ways to close the construction area (both available from the menu): as with TAS, it can anytime be made to reappear as a proof icon in the assembling area, and further, once there are no more subgoals to be proven, it can reappear as a theorem.

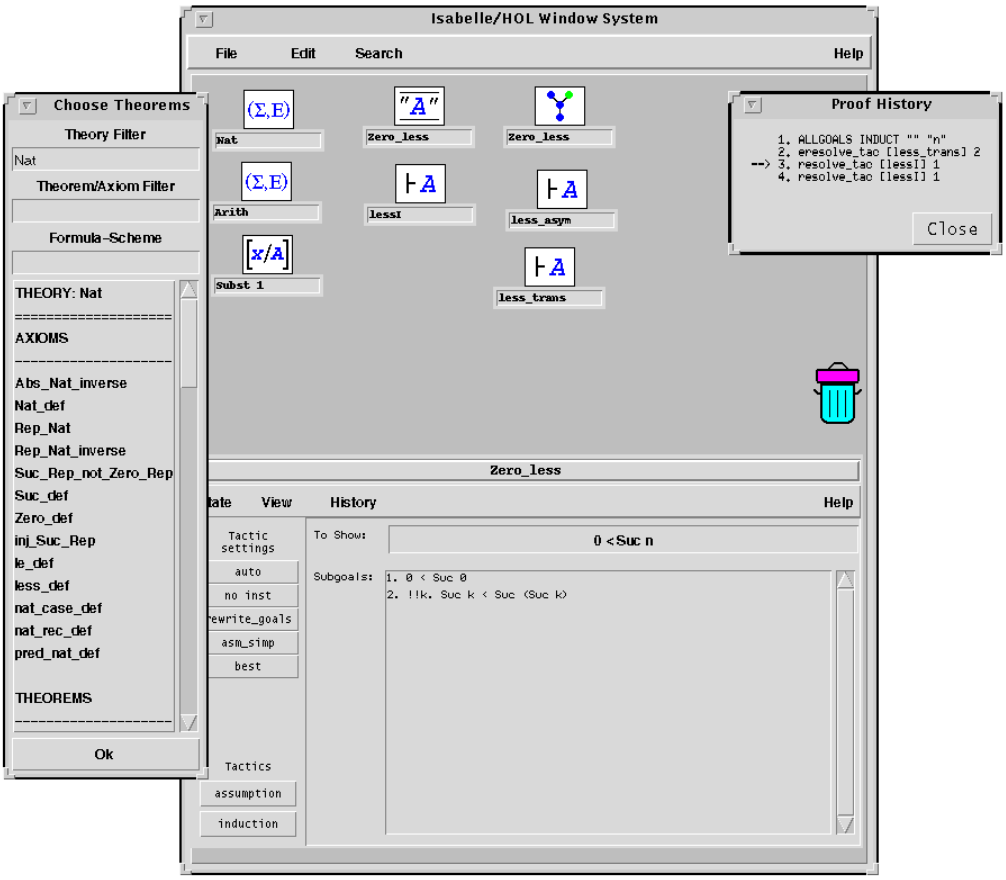

Fig. 3. IsaWin Main Window

5 Related Work, Outlook and Conclusion

5.1 Related Work

Our paper is concerned with two main issues: the design and implementation techniques for formal method tools, and the integration of transformational program development into interactive theorem provers. Related work focuses on either of these issues.

In particular, there is a variety of GUI designs for specific systems like Mathpad [BV96], a tool support for a calculational method, JAPE [BS96] (Just Another Proof Editor) with a "quiet interface", CtCOQ [BB96], an interface to the COQ system and CaDiZ [Toy96], a support tool for reasoning about Z specifications. However, there is not yet an agreed concept of how to build user interfaces for theorem provers (or how they should look like); suggestions include modelling techniques [Mer96] and window inference [Gru92].

Other interfaces to LCF provers like TkHOL [Sym95] and XIsabelle [CO95] are clearly superior to ours (at least for the moment) as far as functionality and user-friendliness is concerned. However, they are predominantly implemented in Tcl and not in ML. We believe that the monolithic design of these systems will be difficult to maintain and extend, and that our approach offers a greater flexibility, leading to a higher extendability and reusability.

The transformational approach has a long tradition, starting from the Munich CIP Project [BBB+85]. During the PROSPECTRA project [HK93], a system has been implemented that enabled the formalisation of transformation rules and their use during the software development process; however, this system was severely hampered by its unstructured design and limited reasoning power, defects which we aimed to remedy by using a powerful prover and a programming language with powerful structuring concepts.

In KIDS [Smi91], programs are developed by transforming *problem specifications* to programs. First, high-level transformations such as global search are used to transform the problem specification to an (inefficient) program which is then optimised by low-level transformations. In KIDS, there is no easy way to check the soundness of the implemented transformations. Here our approach offers a complementary aspect to KIDS since we can prove the correctness of the transformation before applying it, and discharge also the resulting proof obligations in the Isabelle system.

5.2 Future Work

In this section we will sketch a few of the improvements we have in mind for our system.

- *Error Handling and User Guidance:* These two aspects are surely the most important areas in which future work needs to be done.
  There are at least two kinds of errors we can distinguish: First, there are critical errors, such as Isabelle (or the ML interpreter) being unable to parse a theory file or command line. Errors of the first kind have been nearly eliminated by restricting the access to Isabelle. Second, there are non-critical failures, such as

the application of a tactic failing. 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 printing any information that can possibly be useful). This should be best included in the tactical sugar of the transformation, where most knowledge about the failure situation is available.

- *Structured View of the Assembling Area:* When working on a larger problem, the assembling area tends to clutter up with too many objects. For this reason, we intend to provide structuring mechanisms like folders and multiple object collections. Further means to hide objects and folders (possibly depending on the actual state of the development) will be investigated.
- *Object Modes:* When manipulating objects on the assembling area, one often wants to modify the drag&drop-operations by additional information attached to the objects. For instance, one might like to declare a theorem (in the IsaWin-example) as a rewrite-rule, an introduction-rule or an elimination rule, and this information may be taken into account when adding the theorem into a rewrite-set or applying in a proof-state.
- *The Concept of a Focus:* In instances of GenGUI based on Isabelle, we intend to provide a more refined construction area with a view on a structured proof state, the parts of which can be selected individually by pointing on them by the mouse. A selected part is called a *focus*.
  The focus can be used to inform the user what rules are applicable, and to control the application of transformations to a subterm of the current goal, corresponding to application of transformations in context involving automatic application of monotonicity rules.
- *A Generic History Module:* It turned out that both in TAS and in IsaWin, a protocol mechanism for transformations resp. tactics is needed. It should be possible to extract ML tactical programs that allows the replay of the developments independently of the graphical user interface from these protocols. This task can also be realized by a generic package that provides the necessary functionality for instances of GenGUI based on Isabelle.
- *Pretty Printing and Proof Editing:* We envisage a pretty printer which converts the symbols of Isabelle's 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

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

- *Further Instances:* One can conceive instantiations of our system for any formal method which can be modeled within Isabelle, such as CSP and Z, for which we have developed an embedding into Isabelle/HOL[KSW96b]. However, the question of an adequate user interaction model for Z (or CSP) still needs some research, that is what are the relevant objects and operations a user wants to see and manipulate following the Z development methodology.

5.3 Concluding Remarks

We have shown how to implement generic graphical user interfaces for theorem provers and transformation systems. The implementation is performed by using a flexible system architecture. Furthermore, we have exploited the advantages of the Tcl/Tk toolkit, but avoided its disadvantages by encapsulating it into SML.

Based on the expressive SML module system, the generic nature of the system allows the building of a graphical user interface, like TAS, by merely instantiating a functor. By allowing the development of easy-to-use prover applications, we hope this will help to open up and develop new fields for system design of theorem proving environments in the future.

References

[BB96] J. Bertot and Y. Bertot. The CtCoq experience. In N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report, pages 17–24. University of York, 1996.

[BBB+85] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, R. Gnatz, F. Gei selbrech tinger, E. Hansel, B. Krieg-Brückner, A. Laut, T. A. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. The Wide Spectrum Language CIP-L.* LNCS 183. Springer Verlag, 1985.

[BS96] R. Bornat and B. Sufrin. Jape's quiet interface. In N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report, pages 25–34. University of York, 1996.

[BV96] R. Backhouse and R. Verhoeven. Mathpad – tool support for the calculational method. In N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report, pages 9– 16. University of York, 1996.

[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.

[For95] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2*, December 1995.

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

[Gru92] J. Grundy. A window inference tool for refinement. In C. B. Jones, B. T. Denvir, and Roger C. F. Shaw, editors, *Proceedings of the 5th Refinement Workshop*, Workshops in Computing, pages 230–254, Lloyd's Register, London, January 1992. BCS FACS, Springer Verlag.

[HK93] B. Hoffmann and B. Krieg-Brückner. *Program Development by Specification and Transformation.* LNCS 690. Springer Verlag, 1993.

[Hoa96] C. A. R. Hoare. How did software get so reliable without proof? In M. C. Gaudel and J. Woodcock, editors, *Formal Methods Europe*, LNCS 1051, pages 1–17. Springer Verlag, 1996.

[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. `http://www.informatik.uni-bremen.de/~uniform/`

[KSW96a] 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*, LNCS 1051, pages 629– 648. Springer Verlag, 1996.

[KSW96b] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle. In J. von. Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, LNCS 1125, pages 283 – 298. Springer Verlag, 1996.

[LWW96] C. Lüth, S. Westmeier, and B. Wolff. sml_tk: Functional programming for graphical user interfaces. Technical Report 7/96, Universität Bremen, 1996.

[Mer96] N. Merriam. Two modelling approaches applied to user interfaces to theorem proving assistants. In N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report, pages 75–82. University of York, 1996.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Pau91] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Pau94] L. C. Paulson. *Isabelle - A Generic Theorem Prover.* Number 828 in LNCS. 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`.

[Shi96] H. Shi. A semantic matching algorithm: Analysis and implementation. In W. Penczek and A. Szalas, editors, *Proc. Mathematical Foundations of Computer Science '96*, number 1113 in LNCS, pages 517– 528, 1996.

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

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

[Toy96] I. Toyn. Formal reasoning in Z using CADiZ. In N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report, pages 119–125. University of York, 1996.

This article was processed using the LaTeX macro package with LLNCS style