Functional Design and Implementation of Graphical User Interfaces for Theorem Provers

Christoph Lüth ∗
Bremen Institute for Safe Systems, TZI, FB 3, Universität Bremen
cxl@informatik.uni-bremen.de

Burkhart Wolff ∗
Institut für Informatik, Albert-Ludwigs-Universität Freiburg
bu@informatik.uni-freiburg.de

Abstract

The design of theorem provers, especially in the LCF-prover family, has strongly profited from functional programming. This paper attempts to develop a metaphor suited to visualize the LCF-style prover design, and a methodology for the implementation of graphical user interfaces for these provers and encapsulations of formal methods. In this problem domain, particular attention has to be paid to the need to construct a variety of objects, keep track of their interdependencies and provide support for their reconstruction as a consequence of changes. We present a prototypical implementation of a *generic* and *open* interface system architecture, and show how it can be instantiated to an interface for Isabelle, called `IsaWin`, as well as to a tailored tool for transformational program development, called `TAS`.

1 Introduction

The story of graphical user interfaces (GUI's) for theorem provers and formal method tools as a whole is not exactly a success story so far. There is widespread scepticism (Merriam & Harrison, 1997) that GUI's adopting techniques from the field of human computer interaction (HCI) can increase productivity to a similar extent as they did, say, in the area of office applications. GUI's for widely used tools like PVS, FDR or Isabelle are still dominated by text-based subwindows barely hiding the roots of the underlying tool. We believe this has predominantly historic reasons.

The history of functional languages, in particular ML, has been deeply intertwined with the genesis of the LCF theorem prover family, for which it was originally developed as a meta language. The essential idea in LCF-style provers (like HOL (Gordon & Melham, 1993) or Isabelle (Paulson, 1994)) is to encapsulate the logical engine in an abstract datatype, the objects of which can only be constructed

∗ This work has been partially supported by the German Ministry for Education and Research (BMBF) as part of the project UniForM under grant No. FKZ 01 IS 521 B2.

by operations implementing the rules of the underlying logic. This yields the basis for an open system design allowing user-programmed extensions in a logically sound way. The flexibility, generality and expressiveness of LCF-style provers makes them symbolic programming environments, into which other languages can be logically embedded, e.g. Haskell (Regensburger, 1994), Java (Nipkow & von Oheimb, 1998), Z (Bowen & Gordon, 1994; Kolyang, Santen & Wolff, 1996b) or CSP (Tej & Wolff, 1997). Together with appropriate, customised proof support and a graphical user interface which hides the details of the embedding, this leads to an implementation technology for formal method tools which we call *encapsulation*.

Thus, while the LCF-design has its undoubted advantages, these systems have inherited a very restricted model of user interaction based on a command line interface, and not profited as much as possible from recent advances in interface design (Shneiderman, 1998; Thimbleby, 1990; Dix *et al.*, 1998). As Bornat and Sufrin (1998) put it, this problem cannot be overcome by "bolting a bit of Tcl/Tk onto a text-command-driven theorem prover in an afternoon's work".

Our contributions towards filling the gap between classical command-line interaction and more modern concepts of graphical user interaction are the following. First, we develop a *new metaphor* for the visualization of LCF-style provers. The metaphor serves as a vehicle to make the data structure of the prover accessible to *pervasive direct manipulation.* Second, the metaphor develops an abstract notion of user interaction, and is compatible with the need for their *systematic replay.* Replaying proofs is a central issue in theorem proving. Third, the metaphor is implemented in a *generic system architecture*, based on the structuring mechanisms of Standard ML, using a functional encapsulation of Tcl/Tk and the theorem prover Isabelle. Besides a graphical user interface for a theorem prover, this gives an encapsulation technique for formal methods.

This paper is organized as follows: we will first discuss issues relating to the conceptual design and the metaphor. We will then turn to the architecture of the system, introducing a data model and a process model. This will be followed by a section expanding on some aspects of the implementation, and a section introducing a different instantiation of the generic system. We close with an evaluation of the proposed work, and a comparison to related work.

2 Conceptual Design Issues

Direct manipulation, a term attributed to Shneiderman (1982), is a widely known technique in HCI and graphical user interface design (Shneiderman, 1998; Thimbleby, 1990; Dix *et al.*, 1998), characterized by *continuous representation* of the objects and actions of interest with a *meaningful visual metaphor* and *incremental, reversible, syntax-free operations* with *rapid feedback* on all actions. In this section, we will introduce the notepad metaphor, serving as a vehicle to make the internal objects of a theorem prover accessible for direct manipulation.

2.1 The Notepad Metaphor

As a motivating example, consider the way we do everyday mathematics and calculations: one typically uses a piece of paper or a blackboard to write down intermediate results, calculations or lemmas, but overall in an unstructured way, adding a column of numbers in one part of the pad, while doing a multiplication in the lower corner and a difficult diagram-chase in the middle.

A simple instance of this is a small notepad on which we can write down numbers and arbitrary text. The operations would either be arithmetic (e.g. add two numbers), or textual (write some new text). Technically, the notepad could be visualized as a window in which the user can manipulate *objects*, represented as *icons*, by *drag&drop*. The world of objects on our notepad is structured by an inherent notion of *typing* (here, numbers and texts). This typing is crucial when considering the operations, because an operation taking numbers as arguments is different from an operation taking texts as arguments. The operations are applied by drag&drop, so if we drop a number onto a number, we may want to add them up, whereas if we drop a text onto a text, we may want to concatenate them. Figure 1 illustrates our example: On the left, we can see objects representing numbers 2, 4 and 5, and two pieces of text. If we move the number 2 on the number 4 (second from the left), they are added up, and we obtain a new object: the number 6 appears (third from the left).

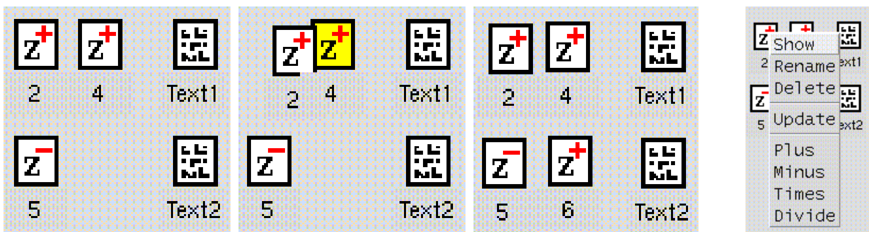

Fig. 1. Introducing the notepad metaphor and manipulation by drag&drop.

This shows the first main principle of a functional GUI: objects represent values, and hence the interaction of objects produces new objects, rather than changing existing ones. Dropping an object onto another corresponds to function application. These functions are called *binary operations*; passing several objects to a binary operation is possible by grouping objects via multiple selections. Additionally, *unary operations* may be defined for each object type, which take exactly one argument, and are invoked via a pop-menu (see Fig. 1 on the right, where the standard operations *Show*, *Rename* and *Delete* can be seen.)

In practice, the simple typing discipline has proven insufficient — e.g. instead of adding two numbers, we might as well want to subtract, multiply or divide them. To this end, we introduce the concept of a *mode* that an object may have. In our example, each object of type natural number has four modes: *plus*, *minus*, *times* and *divide*. The function applied by drag&drop is determined by the mode of the object being dropped onto: dropping a number onto another number in *times* mode multiplies the two numbers.

At first sight the modes seem to contradict our principle of a functional GUI, since they allow a form of state. However, the modes only serve to disambiguate or simplify user interaction. This context information may help the system to provide additional parameters that had to be provided explicitly otherwise; we do not allow side effects when applying operations. As a general rule of interface design(Thimbleby, 1990), modes should not be hidden, so the icon of an object is determined by both the mode and the type of the object. This way, the action which will take place is always transparent to the user. In Fig. 1, the modes of the numbers are shown by an additional sign on the upper right corner of the symbol. The user can change the mode of an object by a pop-up menu (Fig. 1 on the right).

2.2 Undo, Persistence and Replay

According to the main principle of a functional GUI, function application can not change the arguments of the function. This allows an easy implementation of *undo*: we just delete the object created by applying the function. Moreover, the functional approach makes it easy to reconstruct an object value. By recording the operations which have been applied to construct an object, we can reconstruct the object value later by *replaying* the operations. This is needed to implement a persistent state, and to deal with external objects.

By *persistent state*, we mean that we want to be able to save the current state at a given moment, exit the system, and later restart the system in the same state where we left it. Under the assumption that only operations, but not objects, can be saved externally, persistence is achieved by recording for every object how it was constructed, and reconstructing the object by replaying the operations upon restart.

As an example of *external objects*, suppose that the texts on our notepad are given as post-it notes stuck to the pad. Their value is the text written on them. We can concatenate two texts, but if we then write something different on one of the notes, the value of the concatenation should change accordingly. In our example, suppose the text objects (like *Text1* and *Text2* in Fig. 1) are read from external files. By dragging *Text1* on *Text2*, we create a new object, say *Text3*. If now *Text1* is *reloaded*, the value of the object may change, and consequently the value of *Text3* should change as well. We say *Text3* is *outdated*, which is indicated by shading the icon of *Text3*. An outdated object is updated again by replaying its *history*. Updating can be invoked manually via the pop-up menu (as in Fig. 1), or automatically. In many applications, however, automatic replay is inconvenient since it may take a long time, and since it may fail, leading to errors which the user has to correct interactively. This may distract the user from his current task, so we let him postpone the updating until it is convenient.

Replay is very important in the theorem proving context, because most theorem provers read declarations and definitions from external files, which are frequently modified by the user, and have to be reloaded. Yet, systematic replay is to our knowledge never supported on the level of the GUI.

2.3 Construction Objects

For certain objects, manipulation without regard to their internal structure, and hence their history, is insufficient. For instance, we want to admit editing of text objects in our simple example. The history of such editing operations will then consist of a protocol of operations like `delete "javelin" at position 3.12` or `insert "spear" at position 3.12`. Navigating forward and backward in this history corresponds to undoing and replay.

These objects will be called *construction objects.* They can be opened by double-clicking, which leads to the creation of two new windows, namely the *construction area* and the *history navigation window* (see Fig.2). Both windows have a *focus*, i.e. a mark on some position in the text, and some position in the history, respectively. The history focus controls the content of the construction area.

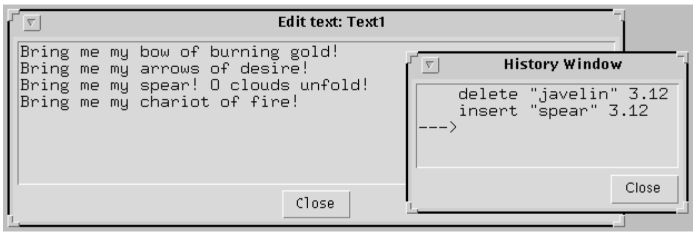

Fig. 2. Construction area and history navigation window.

When closing a construction object, the current value of the construction object is bound to the object that was opened (i.e. to the icon that was double-clicked). The reason for this behaviour is that the notepad would hopelessly clutter up if a new object was created for each step in the history. Objects depending on a closed construction object are marked as outdated.

Note that the replay of the history may fail. For example, the semantics of the `delete "javelin"` operation may be undefined if no `"javelin"` occurs in the text as a consequence of an external change and a reload of the object.

There are more forms of interaction between the notepad window and the construction area or the history window. Objects on the notepad may be dragged on the focus set in the object value field, e.g. replacing the text selected in the focus. *Vice versa*, the selected text of the focus may be extracted and form an object on the notepad.

2.4 `IsaWin` — *A Functional Graphical User Interface for Isabelle*

We will now explain how theorem proving fits into the concepts described in the previous sections by describing our GUI `IsaWin` for the theorem prover Isabelle

(Paulson, 1994). Isabelle is just the example at hand; we expect no principal difficulties in developing analogous interfaces for other LCF-style prover based on SML.

2.4.1 Accommodating Basic Theorem Proving into the Notepad

The object types of `IsaWin` are a subset of those provided by Isabelle, as shown on the left of Fig. 3: in the first row two theorems and a theory, in the second row, two different types of rule sets, called simplifier sets and classical rule sets in Isabelle parlance, and an ongoing proof, or more precisely the proof script which we will identify with a proof throughout this paper. Theorems have four modes: they can be introduction rules (as in Fig. 3), elimination rules, destruction rules and equations (which are not shown).

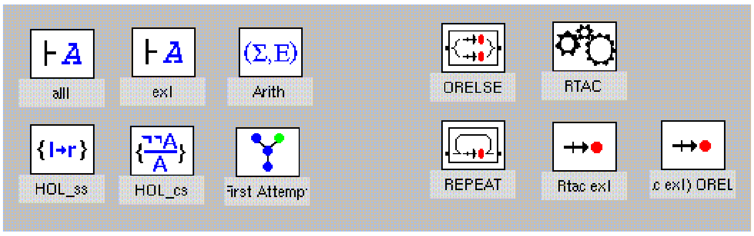

Fig. 3. The Objects of `IsaWin`: to the left, basic objects; to the right, tactical objects

The binary operations in this instance include the forward resolution of two theorems: unifying the conclusion of one theorem with the hypothesis of another one. This corresponds to dropping a theorem object onto another theorem. Note that this may not succeed — the operation is partial. E.g. if the theorem `add_0` : $0 + c = c$ is dropped onto the theorem `sym` : $s = t \Rightarrow t = s$, a new theorem $t = 0 + t$ is produced by forward resolution; but *vice versa* the operation fails. Simplifier sets are sets of rewriting rules. If a theorem is dropped on a simplifier set, a new simplifier set is produced with the theorem included. Classical logics support another type of rule sets. These classical rule sets come in two modes, safe and unsafe, since theorems can be added to a classical rule set in two ways (this distinction makes a difference for a decision procedure of Isabelle, the so-called classical reasoner). If a theorem is dropped on a classical rule set, depending on the mode of the rule set, it is either added as a safe rule or as an unsafe rule.

Reloading an external theory file results in outdating all dependent objects, like included theories or theorems depending on them.

2.4.2 Accommodating Tactical Programming into the Notepad

Contrary to a common prejudice against GUI's for theorem provers, it is quite straightforward to embed basic tactic script construction into the notepad metaphor. First, we provide object types corresponding to certain Isabelle types, *tac_op* (for tactical operations), *tactics* and *tacticals*. Second, we provide objects of type *tac_op* corresponding to backward resolution or simplification, basic tactics like proof by assumption as objects of type *tactic*, and the usual connectives *REPEAT*, *THEN* and *ORELSE* on tactics as objects of type *tactical*. Third, we set up the binary operations by embedding Isabelle's tactical algebra into our world of object types, objects and binary operations. We are now able to construct, for example, an object corresponding to the Isabelle tactic `REPEAT o ((rtac exI) ORELSE' (rtac allI))` which by repeated backward resolution with the quantifier introduction theorems *exI* and *allI* will eliminate an arbitrary sequence of outermost quantifiers on a subgoal. Fig. 3 shows the constructed tactic in the lower left corner, together with tacticals *ORELSE* and *REPEAT*, the tactical operation *RTAC* and the tactic *Rtac exI*.

2.4.3 Accommodating Backward Proof into Construction Objects.

In LCF-style provers, the main proof method is by *backward proof*: if we want to prove a goal ϕ in this style, a *proof state* is initialized with the formula $\phi \Rightarrow \phi$. With a theorem $A \Rightarrow B \Rightarrow \phi$, the proof state can be refined to $A \Rightarrow B \Rightarrow \phi$ by forward resolution. The premises left from the rightmost implication, here A and B, are called *subgoals*. If, as a consequence of further proof steps, no subgoals are left, the proof state can be converted into the theorem ϕ.

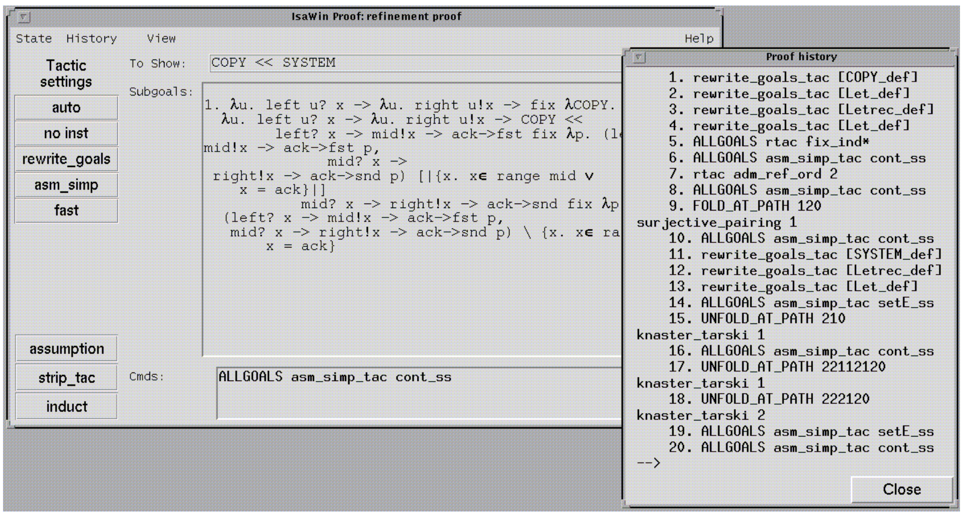

Fig. 4. `IsaWin`'s construction area

It is convenient to declare backward proofs as construction objects and the proof steps performed by tactical operations as their history. When dragging objects from the notepad window to the construction area, the GUI will perform tactical operations, depending on the mode of the dragged object, the settings of the buttons and the focus set by the user. If a theorem `lemma1` has the mode *introduction rule*, and the focus is set to the second subgoal, the drag&drop gesture will trigger the

Isabelle operations `by(rtac lemma1 2)`. Further, dragging a simplifier set down into the construction area will cause Isabelle's rewriting machine to execute the rewrites in it. If there are no subgoals left to be proven, the construction area can be closed to yield a theorem object on the notepad. Figure 4 shows the construction area of the `IsaWin` interface. The most prominent part is the display of the subgoals, and the main goal to be proven.

3 System Design Issues

As mentioned in the introduction, we want to provide a family of user interfaces for different *applications*, let it be for different theorem provers or different tools built on them. Hence, our system architecture has to be *generic*. It depends on an abstract characterisation of the application; this parameter is discussed in Sect. 3.1 and leads to the data model described in Sect. 3.2. This view is complemented by the process model in Sect. 3.3, where the communication of the different components is presented.

3.1 An Abstract View of Functional User Interfaces

At an abstract level, we consider the theorem prover, or the encoded formal method, to be an *application* which is a structure with the following characteristics:

- It has *objects*, each of which has *type*. The type determines the possible *modes*, and both determine which operations are applicable to this object, and both were indicated by the object's *icon*.
- There are partial *operations* which can be applied to objects, namely *unary operations* which take exactly one argument, and *binary operations* which take two arguments. Unary operations are selected from the pop-up menu bound to each object, whereas binary operations are triggered by drag&drop.

Thus, an application has a set S of types, a set Ω of operations which have certain arity (i.e. an operation $\omega \in \Omega$ takes exactly one or two arguments of specific types), a set A_s of the possible values of objects of type s, and a way to apply operations ω from Ω to elements of A_s. In other words, an application is given by a signature $\Sigma = (S, \Omega)$, and a partial Σ-algebra A. This separation of the syntax of the application (given by a signature) from its semantics (given by an algebra) is essential in being able to handle replay, as we will see below.

The modes — and similarly, the settings in the construction area — only serve to disambiguate which operation ω is going to be applied. Once the operation has been selected, its evaluation is independent of modes, settings or any other user input.

Technically, we can denote this characterisation by an SML signature `APPL_SIG`, making the generic interface an SML functor which when instantiated with an application yields a graphical user interface for that application; we will elaborate on this in Sect. 4.2 below.

3.2 The Data Model

The metaphor developed in the previous section was based on the representation of values by icons on a notepad, and application of operations on these objects. Representation on the notepad corresponds to *naming* an object— the object can be referred to, and operations can be applied to it.

As an application is given by a signature Σ and a partial Σ-algebra A, the history of an object is given by composition of operations, or in other words by a *term* t from the term algebra $T_\Sigma(X)$, built over a set X of variables (where the rôle of the variables is taken by the external identifiers). Then, given a mapping of the variables to values in A_s (i.e. a way to evaluate external objects), every term evaluates to an element of A_s (MacLane & Birkhoff, 1967). So in order to be able to replay the construction of an object, every object is represented internally as a pair (a, t) with $a \in A_s, t \in T_\Sigma(X)_s$, where a is the current optional value of the object (if it exists), and t is the history.

Since objects can be referred to, a single object can be used more then once. The data model has to take into account that kind of sharing, since otherwise replay would become unnecessarily expensive. In proof scripts, this sharing is achieved by binding the theorem to an identifier. In our data model, it is implemented by representing all terms representing the history of the objects in a directed acyclic term graph, representing the global data state of the system.

The vertices of the term graph correspond to the pairs (a, t); every vertex may be associated to an icon on the notepad. The edges correspond to operations. If the value a does not exist, the object is called outdated. Recall that outdating can occur in two ways: an external object is changed (i.e. re-evaluated; for example, a file is being reread into the system), or an operation is applied to a construction object.

The notion of history used here is linear, like Archer, Conway and Schneider's script model (Archer *et al.*, 1984). When we go back in the history, there is a sequence of operations which can be applied by going forward again (the pending operations). If, after going back, we apply a different operation, these pending operations are lost and cannot be referred to anymore. This is a design decision to make navigating the history easy. With the data model, it would be easy to implement a history which is not a linear script, but a graph (like Vitter's US&R model (1984)), where applying new operations is possible while still pending ones are kept in another branch of the history.

3.3 The Process Model

The components of the architecture comprise the notepad and the construction area which have already been introduced above. Additionally, the application may provide communicating components such as a file selector, or a theorem chooser; these typically serve to import external objects into the system. Construction area and notepad are closely coupled, because they exchange values of objects under construction. The notepad and all other components communicate with each other via

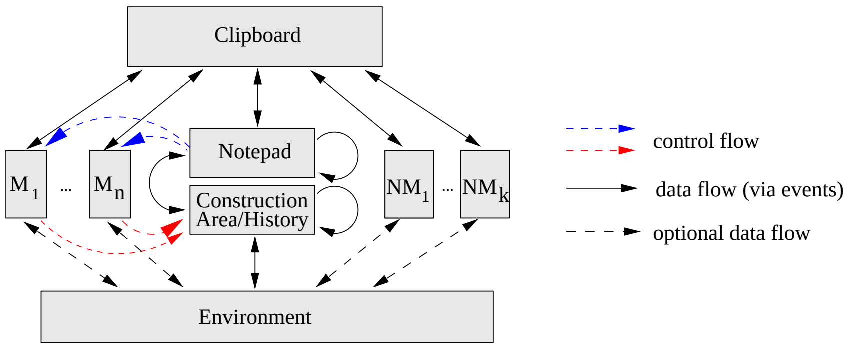

Fig. 5. The Process View of the Architecture Scheme.

a clipboard, and with an external environment, exchanging external representations of the history of objects.

Figure 5 shows the process view of the system. $M_1, \ldots, M_n$ and $NM_1, \ldots, NM_n$ are the application-specific *modal* and *non-modal* components. Whenever a modal component is activated, communication with all other components is refused, while non-modal components allow interleaved communication. Hence, activation of a modal component transfers the control flow of the interface to this component, as indicated by the dotted arrows. Except for the environment and the clipboard, every component is associated to a widget or a window visualizing its process state in the GUI (construction area and history navigation have one each for convenience). The components may optionally communicate with the environment (hence the dashed arrows).

Our design goal of pervasive direct manipulation is reflected by the communication vertices that connect all components with the clipboard. The arrow pointing into the clipboard represent drag-events (parameterized with the object), while the arrow pointing from the clipboard represent drop-events.

Our design goal of persistence is reflected by the arrows connecting the notepad to the environment. Both of these components have an internal state which we have to be able to save into the environment, and read back from there. The application-specific selector components may have an internal state, and thus may need to communicate with the environment as well.

In Fig. 5, more than one instantiation of the interface can be connected, by sharing the same environment, and by connecting the clipboards. This requires conversion functions between the objects of the different instantiations. This gives us a way to build Formal Software Development Environments as a consequence of the genericity of our architecture. A prototypical implementation of this scheme, centred around tools for the specification language Z, is discussed in (Lüth *et al.*, 1998).

4 Implementation

In this section, we will give an overview of the implementation, briefly touching on all components of the system (Fig. 6) in turn. The system is implemented entirely in Standard ML. The instances discussed throughout the paper are based on the theorem prover Isabelle. Since Isabelle essentially consists of a collection of ML types for objects such as theorems, proofs and rule sets, and ML functions to manipulate these objects, organised into a collection of ML structures and functors, one can conservatively extend Isabelle by writing ML functions, using the abstract datatypes provided by Isabelle, without corrupting the logical core of Isabelle.

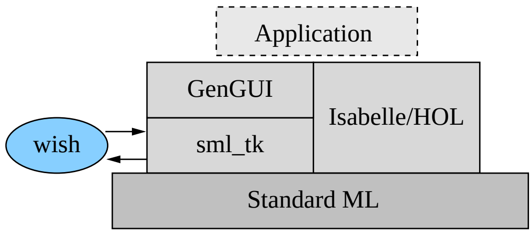

Fig. 6. Module Architecture

To implement the graphical user interface, we have developed a functional encapsulation of the interface description and command language Tcl/Tk (Ousterhout, 1994) into Standard ML, called sml_tk (Lüth *et al.*, 1996). This package provides abstract ML datatypes for the Tcl/Tk objects, thus allowing the programmer to use the interface-building library Tk without having to program the control structures of the application in the untyped, interpretative language Tcl. Further, the sml_tk toolkit library offers a collection of often-used, customisable standard components, such as text-input windows or file choosers.

4.1 Direct Manipulation of Formulas and Annotation Issues

The problem of representing terms and formulas is ubiquitous in a GUI for a theorem prover. With few exceptions based on a dag-like representation (Kahl, 1998), terms are represented essentially text-based, enriched by mathematical or some graphical notation like square roots or sum signs. In a GUI, there is a potential for novel user interaction such as *query-by-pointing* (clicking on a subterm in order to get information like types) or *prove-by-pointing* (clicking on a subterm to apply a tactic or rewrite) (Bertot & Théry, 1998). Finally, direct manipulation is a straightforward idea enabling the user to drag&drop a subterm within a sum, effecting appropriate applications of associativity and commutativity laws (going back to the system *Theorist*; see also (Bertot, 1997a)) which are, at least in Isabelle, extremely tedious to communicate in command-line style.

In a generic, language independent environment such as Isabelle, a prerequisite of theses interactions is the generation of term annotations that allow the user to set a focus in the sense of Sect. 2.3, or to *point* in the sense above. In this section, we describe the necessary concepts and their implementation within the Isabelle syntax engine.

First, a mechanism to attach and manage one or more alternative external representations of a syntax to a theory is needed. These *paraphrasings* allow one to produce graphical output like $\forall x.P$ instead of the conventional text output `!x.P`. (This mechanism also allows the generation of other documentation formats like LaTeX).

Second, for a smooth transition from Isabelle's text-based output to graphical output, we implemented a markup-interpreter as a generic component of sml_tk. It provides a generic parser for an SGML-style notation `<tag> ... </tag>` that binds attributes or ML functions to the subtext marked by the tags; e.g. the graphical output above is obtained from the code `<FONT SYMBOL>\"</FONT>x. P`.

Third, the concept of *annotations* has to be added. Annotations are constant symbols with an external representation that is invisible on the screen and whenever possible transparent to the Isabelle printing macros and printing translations. They are used to generate bindings to specific subtexts. For example, the focus mechanism described above is implemented by surrounding every subterm t with an annotation <SEL p>t</SEL> where p is a representation of the path to the subterm t. The tag `SEL` is bound to a function which, given p, extracts the subterm t from the proofstate. This annotation has to be transparent to the pretty-printing macros; otherwise e.g. the rewriting from the internal representation `x::y::[]` to the external representation `[x,y]` will fail. Based on these paths, it is a standard exercise in tactical programming to provide the necessary operations for query-by-pointing and prove-by-pointing.

Tags and annotations can be nested. For example, in $(\forall x.\, P(x)) \Rightarrow P(t)$, the text x will be annotated as a part of the subterms x, $P(x)$ and $\forall x.\, P(x)$. In such a case, the most specific annotation is selected first, with subsequent clicks cycling through the less specific ones, so above, the first click on x will select the subterm x, the second the subterm $P(x)$, and the third the subterm $\forall x.\, P(x)$.

In summary, a few technical extensions to Isabelle's pretty-printing and parsing machinery are sufficient to make Isabelle support graphical mathematical notation and direct manipulation on terms. These extensions are fully compatible with Isabelle's logical genericity, and fully backwards-compatible with existing syntactic notations.

4.2 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. It is independent of Isabelle, and given as a functor

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

which returns a graphical user interface for the application `appl`. The abstract characterisation of an application has already been introduced in Sect. 3.1 above; we will now give a sketch of the ML signature `APPL_SIG` describing them. The real signature of course is far more elaborate, containing in particular details about the visual appearance (such as the size of the window, or the particulars of the icons depicting the objects and their locations).

The ML signature can roughly be divided into four parts: typing of the objects, operations and applying them, the construction area and external objects.

For the first part, every object has a type given by `obj_type`; its mode can be changed within the modes of the object's type, as given by `modes`. Objects of type `construction_obj` are construction objects, which can be opened and manipulated in the construction area:

```
signature APPL_SIG =
sig
  type object           (* The type of all objects *)
  eqtype objtype        (* The type of object types *)
  eqtype mode           (* The type of modes *)
  val obj_type          : object -> objtype
  val modes             : objtype-> mode list
  val mode_name         : mode-> string
  val initial_mode      : object-> mode
  val construction_obj  : objtype
```

For the second part, there is a type modelling the operations, and an operation with which to apply it. Application is partial, and so the result of an application is either a new object (variant `OK`), or failure (variant `Error`, the string argument is an error message to be displayed):

```
  datatype object_result = OK of object | Error of string
  type opn
  val apply   : opn* object list-> object_result
  val mon_ops : objtype-> ((object* (opn->unit)-> unit)* string) list
  val bin_ops : (objtype* mode)* (objtype* mode)-> opn option
```

For every object type `mon_ops` gives the unary operations as a list of pairs of functions and strings. The string is the name under which the operation will appear in the pop-up menu; the function implements the operation. It gets passed the actual object as its first argument, and a continuation which is used to apply operations. The reason for passing a continuation is that a unary operation may require further user interaction (e.g. when starting a proof in a theory, we first have to enter some goal to be proven).

The binary operations are given by `bin_ops` and come into effect by drag&drop. For every type and mode of a target object (the one being dropped onto) and type and mode of objects being dropped, this function gives an option of an operation; if this option is empty then no operation is available for this drag&drop situation.

The construction area shows the delicate interplay between the application and GenGUI. Because the generic user interface implements the history and commands such as undo, the application cannot provide these. But since the layout of the construction area is given by the application, there needs to be a way to call functions which navigate the history, in order to bind them to graphical control elements. So we model the construction area by a functor, which takes the history navigation functions given by the signature HISTORY_SIG (omitted here), and implements the following export signature:

```
  functor ConArea (structure H : HISTORY_SIG):
    sig val open_area : object*H.history->TkTypes.Widget list
        val drop_ops  : objtype*mode->object list->(opn->unit)->unit
    end
```

The construction area provides the functions open_area which takes an object, and its history, and returns a list of widgets making up the construction area. For every type and mode of an object being dropped from the notepad into the construction area, drop_obs gives the operation to be applied. Like mon_ops, its arguments are the objects and a continuation to allow further user interaction.

The last part of the application deals with external objects. They are referred to by an identifier of type external_id. Given such such a reference, we may obtain an object from that by get_external_obj. The prime example here are file names; get_external_obj loads the contents of the given file. The application may specify dependencies between external objects (see below).

```
  eqtype external_id
  val ext_obj_depends_on : external_id* external_id-> bool
  val get_external_obj   : external_id-> object_result
end
```

The export interface shows a representation of the data model introduced in Sect. 3.2. The type obj_label represents vertices of the term graph. objects is a representation of the term graph as a list of pairs (l, e), where l is a label and e an expression, consisting of applied operations, external objects or references to previous labels.

```
signature GEN_GUI= sig
  type obj_label
  datatype obj_hist = External  of external_id
                    | AppliedOp of opn* obj_hist list
                    | Result    of obj_label
  type objects   = (obj_label* obj_hist) list
  type notepad   = (obj_label* TkTypes.Coord) list
  type gui_state = objects* notepad
  val  change_external_obj : external_id-> unit
end
```

The notepad contains representations of vertices in the term graph on the screen, given as pairs of obj_label and Coord; we only need the coordinates, since the rest of the visual representation will be computed from other information (the type of the object etc.). Then the state of the whole system is given by the term graph and the notepad, and represented by an ML value of type gui_state. Hence we can use the ML parser to restart the interface in a given state, by generating a string which when parsed and evaluated is a value of type `gui_state` corresponding to the current state of the interface. Further, we do not use the whole of ML but only a small subset describing variable declarations `val x=` e and expressions e built by function application; a parser for this subset (or another language with similar expressiveness) would not be hard to implement, allowing to parse and evaluate expressions like above under the control of the interface. This can be used to exchange single objects (e.g. proofs) between sessions, and to integrate a text-based interface into the graphical interface.

Incidentally, the state is represented as a global reference; in Haskell it would be implemented more elegantly as a monad. We do not show the functions used to control the start and restart of the GenGUI, but importantly there is a function `change_external_id` by which an external application can signal that the value of an external object has changed. GenGUI then reevaluates the corresponding external object, and all those external objects which depend on it (as specified by `ext_obj_depends_on`), and moreover outdates all objects constructed from these external objects.

Note how the functional nature of the interface is reflected in the typing: all operations, given by `mon_ops`, `bin_ops` and `drop_ops`, can only produce new objects. The application cannot delete objects.

As a final detail, the clipboard is implemented by sharing a common structure `CLIPBOARD`, which exports two functions, `get: unit-> obj_hist` and `put: obj_hist-> unit`; if (and only if) `put` is called with the history of an object, the next call to `get` will return this history.

5 A Different Instantiation of the Generic Architecture

In this section we will demonstrate how instantiations of our generic architecture can be used to build a special purpose tool by encapsulating a formal method into Isabelle. The tool will be the transformation system `TAS`, similar in spirit to window inferencing (Grundy, 1991) as realized, for example, in the system TkWinHOL (Långbacka *et al.*, 1995), and related to systems such as Prospectra (Hoffmann & Krieg-Brückner, 1993).

5.1 Concepts of `TAS`

In this section, we will briefly sketch the basic principles of modelling transformational program developments in an LCF-style prover, following the lines of Kolyang, Santen and Wolff (1996a). A transformational development can be described as a sequence of correctness-preserving refinement steps

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

One can abstractly view the SP_i as arbitrary formulae and $\rightsquigarrow$ as a transitive, reflexive and monotone refinement relation; this can be, for example, the implication from right to left in the case of refinements based on standard model inclusion, or process refinement as in CSP. Every development step $SP_i \rightsquigarrow SP_{i+1}$ is given by applying *transformation rules*, 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* (Smith & Lowry, 1990).

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*, driving the concrete application in a development context. A logical core theorem has the following general 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*. By proving the logical core theorem, a transformation is proven correct. When applying a transformation, the applicability conditions result in *proof obligations* which are proven externally, by other interfaces to Isabelle (like `IsaWin`) or by decision procedures or (e.g. model-checkers).

The Transformation Application System is designed to hide this implementation in the prover from the user. Since the proof obligations can be deferred to a later stage, the user of a transformation system can concentrate on the main design decisions of transformational program development: which transformation to apply, and how to instantiate its parameters.

5.2 `TAS` *as an Instantiation of the Generic GUI*

We will show how to set up `TAS` as an application in the sense of Sect. 3.1 above. We have to define construction objects, object types, and operations. The construction objects of `TAS` will be transformational program developments, corresponding to Isabelle's proof state, with a history of the transformation rules which have been applied. The object types are transformational program developments, transformation rules with none, some or all of their parameters instantiated, parameter instantiations, texts, and theories. Since in realistic transformation rules (such as Global Search) parameter instantiations are lengthy, instantiations merit an additional dedicated object type to avoid retyping, and to allow copying them.

Fig. 7 shows a screen shot of `TAS` with some objects on the notepad, and a transformational development currently open in the construction area. The operations include instantiating a transformation rule by dropping an instantiation on a transformation rule, and applying a transformation rule by dragging it into the construction area. Further, two transformation rules can be composed (using the transitivity of $\rightsquigarrow$) by dropping a transformation rule onto another one; if present, the application conditions of both transformations were conjoined and the parameters universally quantified again.

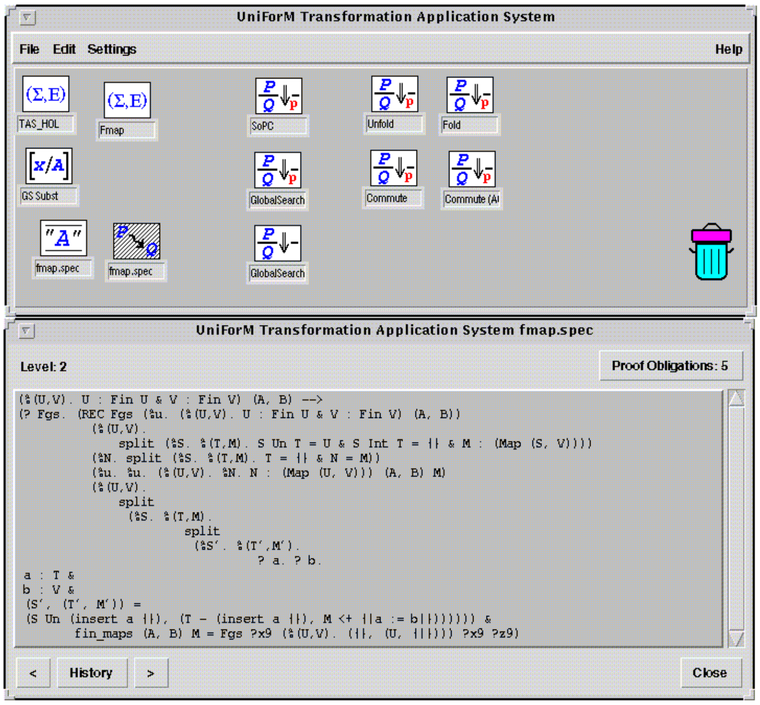

Fig. 7. The Graphical User Interface of `TAS`

6 Evaluation, Related Work and Conclusions

In this final section, we will discuss a metric evaluation of `IsaWin`, briefly review related work and close with a summary of our results and an outlook on future work.

6.1 Evaluation of `IsaWin`

Card, Moran, and Newell (1983) proposed the *goals*, *operators*, *methods* and *selection rules* (GOMS) model and related it to the *keystroke-level model* (KLM). They postulate that the users formulate goals (e.g. prove lemma) and subgoals (e.g. push operator outermost) which they achieve by using methods (press key, move mouse, recall theorem name, etc.). The selection rules are the control structures for choosing among several methods available for accomplishing a goal — statistical assumptions about the deviation of these choices form the basis for a translation into the keystroke-level model. KLM attempts to predict performance times for error-free expert performance of tasks by summing up the time for key-stroking, pointing, drawing, thinking, and waiting for the system. Kieras and Polson (1985), and Elkerton and Palmiter (1991) refined the approach.

The original model, but to a lesser extent also its successors, "concentrate on expert users and error-free performance, and place less emphasis on learning, problem solving, error handling, subjective satisfaction and retention" (Shneiderman, 1998).

Given these fundamental reservations, it is not clear that the GOMS model and its variants apply to theorem proving. Of course, we can do a rough comparison on the KLM-level of `IsaWin` and Isabelle's command-line interface — for example, the proof script in Fig. 4 is generated by 47 elementary user interactions like *set focus* or *drag object*, substantially less than the interaction necessary to produce a proof script of 233 characters — but one might argue that the *fewer interactions* required by a GUI contrast to a larger number of *more usual interactions* (keystrokes) in the command-line interface. And even if most Isabelle experts agree that the proof script shown in Fig. 4 is typical, the question will remain how costly are untypical situations, where the expert user can use the full flexibility of ML. Hence, these metric data ignore factors like subjective satisfaction and usability.

However, we can identify two areas which `IsaWin` handles better than a command-line interface: Firstly, proof-by-pointing and query-by-pointing, allowing by a single mouse-click what in the command-line interface requires the tedious and extremely error-prone construction of substitutions, and secondly, `IsaWin`'s replay, allowing a much finer analysis of which proofs are affected by a change than the conventional rerunning of scripts which fails at the first problem.

In summary, at present claims like "GUI's improve productivity over command-line interfaces in some formal method" can not be founded on taxonomic data, although some first studies (Jackson, 1997) suggest this, at least for a particular prover and GUI. It may actually be the case that a GUI precisely because it is easier to use does not encourage purposeful planning to the extent which is necessary for the successful use of a theorem prover (Merriam & Harrison, 1997). Then again, it may be that a GUI makes the alternatives the user faces clearer and easier to invoke (Bornat & Sufrin, 1998). The question remains open until more systematic studies have been conducted; for `IsaWin`, the prototypical status of the implementation has until now precluded such studies.

6.2 Related Work

6.2.1 Generic Architectures and Abstract GUI Descriptions

Design patterns have recently received a lot of attention in the field of object-oriented programming (Gamma *et al.*, 1990; Cooper, 1998). Also motivated by reusability, some techniques (e.g. templates roughly corresponding to functors) are similar to our generic system architecture. However, important aspects of these patterns are described completely informally, resulting in a sometimes intransparent mixture of meta-language, `C++` code and pragmatics. In contrast, work on "architecture styles" (Abowd *et al.*, 1993; Allen & Garlan, 1994) aims at a fully formal description of generic architectures. However, for the moment, the emphasis of this research lays on foundation, description and analysis and less on implementation. Hence, we consider this work as complementary.

In the HCI literature, there is a large body of work applying formal methods, for the modelling of GUI's, based on temporal logic, Z or process algebras; see (Dix *et al.*, 1998) for a survey. Interface components can be described as processes exchanging events in a process algebra like CSP pp 320. A similar modelling in CSP could be done for our generic system architecture; then even the dialogue behaviour of the application can be described and specified formally.

6.2.2 GUI's for Theorem Provers

GUI's for computer algebra systems such as Maple, Mathematica or MuPad all offer mathematical editing facilities and some of them even direct manipulation of formulae (e.g. rewrite by drag&drop). Typically, this kind of direct manipulation is only available without genericity. These systems are built for a fixed syntax (with emphasis on arithmetics or differential equations), a fixed logic and on the basis of a non-generic system architecture. This also holds for the special purpose theorem prover CADiZ (Toyn, 1996).

In contrast, most recent theorem proving environments are generic, and some also offer proof support for direct manipulation. Jape (Bornat & Sufrin, 1996) is generic in the logic and offers an interface with different styles of proof layout, graphical pretty-printing, and supports proof by direct manipulation, so-called "gestures". It is a lightweight prover, which has not been used yet to encapsulate a formal method. Jape's gestures are similar to CtCoq (Bertot & Bertot, 1996), where they are called proof-by-pointing, but the basic idea remains the same. CtCoq is based on a powerful prover, Coq, which unlike Isabelle is not generic, and moreover supports graphical output which can be configured by the user at runtime, script-based replay, and further direct manipulation like rewriting by drag&drop.

CtCoq is actually part of a larger initiative, in spirit similar to ours, to provide generic interfaces for a family of provers (Bertot & Théry, 1998). The generic interface is implemented using the Centaur system (Borras *et al.*, 1988). In contrast to our architecture, the system is distributed (prover and interface can run on different machines) and heterogeneous (prover and interface need not be implemented in the same language). This work has been taken up by the Proof General project at the University of Edinburgh (Bertot *et al.*, 1997), in which a family of interfaces for the three provers Coq, Lego and Isabelle has been implemented inside XEmacs. In our view, despite practical advantages, this does not lead to a better system architecture; and the close interaction between interface and prover possible because both are implemented in the same language leads to better support of direct manipulation and, in particular, replay.

6.3 Results

In this paper, we have demonstrated how ideas of functional programming applied to user interface design gives rise to a new *functional* visualization metaphor, the notepad. The metaphor serves as vehicle to make the data structures of these provers accessible to pervasive direct manipulation.

The notepad allows for abstract manipulation of *objects* (consisting of *construction history* and an optional *value*) represented by *icons.* The functional paradigm is a precondition for systematic replay, based on the recorded construction history

Module	Code size (lines of SML)
sml_tk	9900
GenGUI	2600
IsaWin	4800
TAS[a]	4500

[a] TAS and IsaWin share about 1400 lines of code.

Table 1. *Size of Code*

of every object. Objects come in two flavours: while standard objects only admit coarse-grained user interaction via drag&drop on the notepad, *construction objects* allow fine-grained user interaction in the *construction area*.

All these concepts are implemented in a *generic system architecture*, based on the powerful modularization concepts of the typed functional language Standard ML. We have presented two instantiations of this architecture, the interface `IsaWin` for the theorem prover Isabelle, and the transformation system `TAS`. As a consequence, we expect that our interface components can be reused for a certain range of similar applications. In particular, this gives a blueprint for the construction of tools with a graphical user interface for formal methods encoded into a theorem prover. We have in turn instantiated `TAS` with CSP and Z, two prominent formal methods for which encodings into Isabelle have been developed.

A fundamental design decision in the implementation was to use the Tk toolkit, encapsulated into Standard ML by sml_tk. The encapsulation sml_tk helped us to survive the evolution of Tk in the recent years while taking advantage of its portability. As Table 1 shows, sml_tk is the largest chunk of code. Building on that, TAS and IsaWin can be kept fairly compact. To put these statistics into context, pure Isabelle has about 17500 lines of ML code.

Despite the relatively low bandwidth between the SML process and the wish, response times have proved satisfactory. The problems arising are minor technicalities: for example, sml_tk builds widgets on the screen by generating and sending Tcl code one line at a time. This results in the interface being incrementally built on the screen, which looks unpleasant, in particular on slower machines. First experiments with generating and sending the Tcl code *in toto* suggest this behaviour can be remedied.

We also have not seen any evidence of Tcl/Tk performance problems, which may be due to the fact that the Tcl code generated by sml_tk is very schematic, with little data handling and control flow. The memory requirements of the interface itself are fairly modest (about 10 MB with the Standard ML of New Jersey compiler, on a Sun UltraSPARC running Solaris 2.6), compared to Isabelle (at least 24 MB, rising to e.g. 33 MB for the CSP encoding). The wish is even more modest, with around 800 KB process size.

In a restricted area of interaction with Isabelle, our instantiations seem to substantially facilitate user interaction. This holds in particular for point-and-query, point-and-prove interactions and for global replay activities.

6.4 Future Work

`TAS` and `IsaWin` are prototypical user interfaces that still need work in details. We would like to allow cut-copy-paste manipulation of the history; in particular the conversion of selected parts of the history to tactic objects would pave the way for powerful techniques of interactive reuse. Further, goals and substitutions are presently read as standard text and parsed via Isabelle's parsing machinery. This should be extended by a suitable mixture with structure-oriented editing facilities as in CtCoq, or mouse-supported input as in JAPE.

As text-based interfaces have their advantages as well, a significant potential for increase in productivity is the integration of a command-line interface into our GUI. Conceptually, the commands and operations which are evoked by the GUI can be expressed in a simple functional language, e.g. a subset of ML (see Sect. 4.2). Hence, a command-line interface offers just a different view of the same underlying behaviour; we merely need to be able to parse and print commands in this language. This can be fully integrated with the rest of the interface; e.g. one could edit and reevaluate commands from the history, or one could provide a function's arguments by dragging their icons down from the notepad.

A far more involved subject is to scale up the systematic *replay* towards automatic *reuse* of former proof attempts. The most evolved replay and reuse techniques we are aware of are realized in the KIV-system (Reif *et al.*, 1997). KIV also provides direct manipulation on the history and moreover automatic support of reuse by detecting unaffected subparts of the proof which can still be used after failed replay. The authors claim that the productivity of this system is essentially due to its reuse techniques (Reif & Stenzel, 1992). However, this feature is based on a very specialized logic. Extending it for a generic theorem prover on the one hand and embedding it into our generic notion of history will represent a substantial challenge, but we believe that the deep incorporation of history both on the system level and on the generic interface level provides a good starting point.

Acknowledgements. We would like to thank the two anonymous referees whose comments helped to improve this paper significantly. Further, sml_tk owes much to GoferTk, developed by T. Vullinghs, D. Tuijnman and W. Schulte at the University of Ulm. Finally, we would like to thank Stefan Westmeier, Kolyang and Thomas Meier for improvements of sml_tk and and fruitful discussions.

References

Abowd, G., Allen, R., & Garlan, D. (1993). Using style to understand descriptions of software architecture. *Proceedings ACM SIGSOFT'93.* ACM Press.

Allen, R., & Garlan, D. (1994). Formalizing architectural connection. *Proceedings of 16th conference on software engineering.* ACM Press.

Archer, J. E., Conway, R., & Schneider, F. B. (1984). User recovery and reversal in interactive systems. *ACM trans. on programming languages and systems*, **6**(1), 1– 10.

Bertot, J., & Bertot, Y. (1996). The CtCoq experience. *In:* (Merriam, 1996).

Bertot, Y. (1997a). Direct manipulation of algebraic formulae in interactive proof systems. *In:* (Bertot, 1997b).

Bertot, Y. (ed). (1997b). *User interfaces for theorem provers UITP'97*. INRIA Sophia Antipolis. Electronic proceedings at `http://www.inria.fr/croap/events/uitp97-papers.html`.

Bertot, Y., & Théry, L. (1998). A generic approach to building user interfaces for theorem provers. *Journal for symbolic computation*, **25**(2), 161– 194.

Bertot, Y., Kleymann, T., & Sequeira, D. (1997). *Implementing proof by pointing without a structure editor*. LFCS Report Series ECS-LFCS-97-368. LFCS, University of Edinburgh. See also the Proof General home page at `http://www.dcs.ed.ac.uk/~proofgen/`.

Bornat, R., & Sufrin, B. (1996). Jape's quiet interface. *In:* (Merriam, 1996).

Bornat, R., & Sufrin, B. (1998). Using gestures to disambiguate unification. *User interfaces for theorem provers UITP'98*.

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., & Pascual, V. (1988). Centaur: the system. *3rd symposion on software development environments*. (also as INRIA Report No. 777).

Bowen, J. P., & Gordon, M. J. C. (1994). Z and HOL. *Pages 141– 167 of:* Bowen, J. .P., & Hall, J. A. (eds), *Z users workshop*. Workshops in Computing. Springer.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human-computer interaction*. Lawrence Erlbaum Associates.

Cooper, J. W. (1998). Using design patterns. *Comm. of the ACM*, **41**(6), 65– 68.

Dix, A., Finley, J., Abowd, G., & Beale, R. (1998). *Human-computer interaction*. Prentice-Hall.

Elkerton, J., & Palmiter, S. (1991). Designing help using a GOMS model: an information retrieval evaluation. *Human factors*, **33**(2), 185– 204.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1990). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Gordon, M. J. C., & Melham, T. M. (1993). *Introduction to HOL: A theorem proving environment for higher order logics*. Cambridge University Press.

Grundy, J. (1991). Window inference with the HOL system. M. Archer, J. J. Joyce, Leveitt, K. N., & Windley, P. J. (eds), *International workshop on the HOL theorem proving system and its applications*. IEEE Computer Society Press.

Hoffmann, B., & Krieg-Brückner, B. (1993). *PROSPECTRA: program development by specification and transformation*. LNCS, no. 690. Springer.

Jackson, M. (1997). A pilot study of an automated theorem prover. *In:* (Bertot, 1997b).

Kahl, W. 1998 (Feb.). *The Higher Order Programming System — user manual for HOPS*. Universität der Bundeswehr München. URL `http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/`.

Kieras, D. E., & Polson, P. G. (1985). An approach to the formal analysis of user complexity. *International journal of man-machine studies*, **22**, 365– 394.

Kolyang, Santen, T., & Wolff, B. (1996a). Correct and user-friendly implementations of transformation systems. *Pages 629– 648 of:* Gaudel, M. C., & Woodcock, J. (eds), *Formal methods europe FME'96*. LNCS, no. 1051. Springer.

Kolyang, Santen, T., & Wolff, B. (1996b). A structure preserving encoding of Z in Isabelle.

Pages 283 – 298 of: von. Wright, J., Grundy, J., & Harrison, J. (eds), *Theorem proving in higher order logics*. LNCS, no. 1125. Springer.

Långbacka, T., Ruksenas, R., & v. Wright, J. (1995). TkWinHOL: A tool for doing window-inferencing in HOL. *Pages 245– 260 of: Higher order logic theorem proving and its applications*. LNCS, no. 971. Springer.

Lüth, C., Westmeier, S., & Wolff, B. (1996). *sml_tk: Functional programming for graphical user interfaces*. Tech. rept. 7/96. Universität Bremen. See also the sml_tk home page at `http://www.informatik.uni-bremen.de/~cxl/sml_tk/`.

Lüth, C., Karlsen, E. W., Kolyang, Westmeier, S., & Wolff, B. (1998). Hol-Z in the UniForM-workbench – a case study in tool integration for Z. *Pages 116–134 of:* Bowen, J. P., Fett, A., & Hinchey, M. G. (eds), *ZUM'98: The Z formal specification notation*. LNCS, no. 1493. Springer.

MacLane, S., & Birkhoff, G. (1967). *Algebra*. New York: Macmillan.

Merriam, N. (ed). (1996). *User interfaces for theorem provers UITP '96*. Technical Report. University of York. Electronic proceedings available at `http://www.cs.york.ac.uk/~nam/uitp96/proceedings.html`.

Merriam, N., & Harrison, M. (1997). What is wrong with GUIs for theorem provers? *In:* (Bertot, 1997b).

Nipkow, Tobias, & von Oheimb, David. (1998). Java$_{\ell ight}$ is type-safe — definitely. *Pages 161–170 of: Proc. 25th ACM symp. principles of programming languages*. ACM Press.

Ousterhout, J. K. (1994). *Tcl and the Tk toolkit*. Addison-Wesley.

Paulson, L. C. (1994). *Isabelle - a generic theorem prover*. LNCS, no. 828. Springer.

Regensburger, F. (1994). *HOLCF: Eine konservative Einbettung von LCF in HOL*. Ph.D. thesis, Technische Universität München.

Reif, W., & Stenzel, K. (1992). *Reuse of proofs in software verification*. Technischer Bericht 26/92. Universität Karlsruhe, Fachbereich Informatik.

Reif, W., Schellhorn, G., & Stenzel, K. (1997). Proving system correctness with KIV. *Pages 859– 862 of:* Bidoit, M., & Dauchet, M. (eds), *TAPSOFT '97: Theory and practice of software development*. LNCS, no. 1214. Springer.

Shneiderman, B. (1982). The future of interactive systems and the emergence of direct manipulation. *Behaviour and information technology*, **1**(3), 237– 256.

Shneiderman, B. (1998). *Designing the user interface*. 3rd edn. Addison-Wesley.

Smith, D. R., & Lowry, M. R. (1990). Algorithm theories and design tactics. *Science of computer programming*, **14**, 305– 321.

Tej, H., & Wolff, B. (1997). A corrected failure-divergence model for CSP in Isabelle/HOL. *Pages 318–337 of:* Fitzgerald, J., Jones, C. B., & Lucas, P. (eds), *Formal Methods Europe FME '97*. LNCS, no. 1313. Springer.

Thimbleby, H. (1990). *User interface design*. ACM Press Frontier Series. Addison-Wesley.

Toyn, I. (1996). Formal reasoning in Z using CADiZ. *In:* (Merriam, 1996).

Vitter, J. S. (1984). US&R: A new framework for redoing. *IEEE software*, **1**(4), 39– 52.