4

What You See is What You Store: Database-Driven Interfaces

J. Thamm and L. Wegner
Universität Gh Kassel, FB 17, D-34109 Kassel, Germany
{injt, wegner}@db.informatik.uni-kassel.de

Abstract

Any graphical user interface (GUI) requires large amounts of complex metadata for the layout of windows and menus, their style and behavior, their bindings, etc. Designing, debugging and maintaining this interfaces is a difficult task and object-orientation is of little help as it tends to overcrowd the code with these metadata. Following a similar idea by Goyal et al., who considered GUI programming a database research problem and gave solutions based on logical programming, we propose storing these metadata in a database which we connect to a scripting language, here Tcl. We explain the details first with a small example where the database acts as a front-end to Mathematica®. Next we indicate schemes, tables and generic methods for automated GUI design. Finally, we discuss the ultimate step, namely managing the interface at run-time from the database. In concluding, we present arguments for our belief that this technique goes well past current GUI builders and has great potential for simplifying design and maintenance of any type of application interface.

1 LIBERATING GUI'S FROM THE DATA-INTENSIVE STYLE

Visualizing data has two aspects. Firstly, visualization will usually be imbedded into a graphical interface made up from a collection of widgets. Through the interface the user can browse, query, and update the data. Today, the design of these conventional aspects of interfaces is supported by interface builders like e.g. Tcl's XF tool (Ousterhout, 1994).

Visual Database Systems 4 Y. Ioannidis & W. Klas (Eds.)
© 1998 IFIP. Published by Chapman & Hall

The second, more difficult, aspect is visualizing the data proper, i.e. translating numeric and textual data into visually intuitive, appealing graphical representations. This task also includes transformation, incorporation, and indexing of other multimedia data, like graphs and pixel pictures, video and audio data, etc. When the target is a standard graphical representation, filters might already exist. Otherwise, new methods must be programmed and tied to the previously mentioned widgets.

Both aspects of the job make visualization difficult and time consuming, mostly because of the complex and idiosyncratic nature of windowing systems. This is true regardless of the actual interface system, whether it is low-level X11, higher level Motif (Brain, 1992), a scripting language like Tcl (Ousterhout, 1994) or Java (Sun, 1995), or VRML programming (VAG, 1996).

Object-orientation has facilitated the task somewhat. In particular, the message passing paradigm is beneficial because of the asynchronous nature of interfaces. Ted Lewis calls this new-era software (Lewis, 1996): "... the software design paradigm changes to a reactive form in which the program responds to events through event handlers. These event handlers are small pieces of code, possibly applets, that perform very specialized and limited functions". However, debugging event handling is rather difficult because of this asynchronous nature.

Secondly, object-orientation implies inheritance and the resulting class hierarchies help in deriving new widgets from existing ones and ordering them into a tree. However, these class libraries have become quite huge and understanding the universe of types puts a heavy burden on those who have to implement the interface and accidental over-writing of existing methods is not uncommon. Also, using pre-existing building blocks still requires good programming skill whereas Ted Lewis foresees a decoupling of software consumption and production in the future.

Finally, object-orientation includes the idea of encapsulation which hides implementation details inside procedures offering to the outside only a limited number of methods (access and modification procedures) for indirectly manipulating the data which also reside inside the procedures. In the case of graphical user interfaces and visualization, however, complex objects are quite often simple records (tuples) with, say 10, components (fields, attributes) and, in the extreme case, ten manipulation and ten query functions all differently named after the fields. Setting and getting parameters for the generic functions of the Xt-library (XtSetArg, XtSetValues, XtGetValues) and the way logical resources like the Graphics Context are handled in X, Xt and Xm come to mind.

The point we argue here is that object-oriented systems have a tendency to turn data into program code and even interactive interface builders like XCESSORY for OSF/Motif or XF for Tcl/Tk generate primarily code and to a lesser extent resource files. When resource files are generated and used at run-time, e.g. the X Window `.Xdefaults` file, they are often in a very simple format and serve only for loading into a main memory record at system start time.

The same tendency is observable in the Web where elaborate styles and demand for interaction and animation have moved more and more metadata (markup instructions, calls of applets, ...) into the pages without a clear separation of both.

The alternative which we propose to change this unsatisfactory design process is

- to store resource data in a suitable database,
- to connect the database to the visualization engine either during the design stage or even at run-time.

Similar ideas appear in an area called *active databases* or *active object systems* (Buchmann, 1994), e.g. with procedures as data types as a feature of POSTGRES (Rowe and Stonebraker, 1987), trigger systems, as proposed for SQL3 (Date and Darwen, 1993) or rule systems, like the ECA-rule system (Dayal et al., 1988).

The remainder of the paper is organized as follows. In Section 2 we argue in favor of an object-relational DBMS as a platform for visualization and introduce our visual database editor ESCHER. Section 3 discusses ESCHER's interaction mode which has been described before (Wegner, 1989; Wegner et al., 1996b). Section 4 then proposes navigational extensions to Tcl, called TclDB, modelled after ESCHER's interaction mode. They were initially suggested in (Wilke et al., 1997) and are now operational. Section 5 gives a short example of the use of skripts in connecting our database editor to Mathematica.

Sections 6 and 7 then carry the idea from a database front-end for visualization tools to interface design in general and discuss the advantages of this approach. They take an outlook at what is needed to fully include event handling in our approach, i.e. to run the interface from the database itself rather than from a code translation of the database contents. Section 8 concludes the paper with a summary and observations concerning efficiency and performance.

2 INTRODUCING ESCHER

There is agreement that "plain" relational DBMS cannot support demanding graphical applications like e.g. CAD-system front-ends. This stems from the need of a RDBMS to normalize data and to reconstruct complex hierarchies through expensive joins. Secondly, lack of suitable types with respect to multimedia applications used to be another argument against relational databases.

By now, RDBMS have been extended to include complex objects (at least structurally if not behaviorally) and a richer set of basic types. They are often called object-relational DBMS and we shall use one particular variant, namely nested relational databases, for our demonstrations. Note that the DBMS must provide a powerful interactive browser and editor because the interface design and later debugging stages require visual support. Our prototype editor ESCHER (Wegner, 1989, Thamm et al., 1996) provides this support as can be seen in Figure 1 where the displayed data are packing information for widgets from a Tk demo. Details of ESCHER and of the example are given below.

A natural alternative to the object-relational model would be one of the available fully object-oriented DBMS (OODBMS). However, if these are derived from object-oriented languages augmented by persistence and transactions, nothing is gained because we end up again with "compiled solutions".

TABLE: Tcl_pack.tbl / SCHEMA: Tcl_pack.scm / APPL: widget

File Edit Fingers Query Sorting Options Help

{}pack infos			
#atom	path	<>options	
		name	value
?d?	.button.b3	pady	2
?d?	.button.b4	expand side pady	yes top 2
?d?	.button.buttons	side fill pady	bottom x 2m
?d?	.button.buttons.code	expand side	yes left
?d?	.button.buttons.dismiss	expand side	yes left
?d?	.button.msg	side	top
?d?	.check.b1	anchor side pady	w top 2

Null OK Cancel

Mode: Browse Active Finger: F1

In Out Prev Next BeginQ Save

Figure 1: Pack information as NF^2 table with two fingers.

The same argument made us also look for interpreted (scripting) languages as connection between database system and visualization engine. Java (Sun, 1995) and Tcl (Ousterhout, 1994) are two obvious choices and we opted for Tcl mostly for historical reasons. Tcl is then extended to permit navigational access to the database (see Section 4 below) because ESCHER provides a cursor-based mode for browsing and editing tables. However, other interconnection modes, e.g. embedded SQL, are conceivable with an object-relational DBMS and in fact ESCHER provides declarative query facility using a QBE-like interface (Wegner et al., 1996a).

The resulting scripts may then be stored as well inside the object-relational database system. Given suitable tables with resource data (Section 7), scripts and tables together effectively implement the graphical interface of the very same database system which stores them. This self-referential nature is a fundamental construction principle of our prototype and should explain why it is called ESCHER after the famous Dutch artist *M.C.Escher* (1898-1972) whose drawings feature the impossible self-referential worlds.

3 A SHORT GLIMPSE AT ESCHER'S INTERACTION MODE

ESCHER supports visualization in non-standard applications and serves as a research platform into areas such as multimedia and visual information systems (Thamm et al., 1996), QBE-like queries (Wegner et al., 1996a) and computer-supported cooperative work (CSCW) (Wegner et al., 1996b). ESCHER is available as public domain software from our ftp site in Kassel.

Here, we are mostly concerned with GUI-development aspects, thus we refrain from going into the data model aspects and concentrate on the interface implementation principles.

Figure 1 shows a window over a table with packing information for a demo widget. The table can be edited using cut-and-paste with a clipboard. As can also be seen, a *schema* is displayed above the *table*. The schema defines packing information, i.e. pack info is a set-valued attribute taking tuples with 3 attributes as members: a numeric atom field, here all set to null (we explain below why there are no ids entered), a string valued path to the widget whose packing info we give, and a list (ordered collection) of options, which in turn are tuples with name and value attributes, both string-valued.

As a visual database editor, ESCHER supports browsing and navigation by means of so called *fingers*. They generalize the cursor paradigm in graphical and text editors. On the graphical display, a finger is reflected by a colored area which corresponds to the object a finger is currently pointing at. In a table, as in Figure 1, more than one finger may point to objects, one of which is the active finger and is used for navigating through the table.

Essential operations on fingers are the navigational operations "going into an object" (*In*, i.e. descending into the next deeper nesting level), "out to the surrounding object" (*Out*), "to the next object" (*Next*, staying on the same nesting level), and "to the previous object" (*Prev*, *Back*). At the interface, the user navigates through the instances using these basic finger operations, either by clicking on buttons in the lower left corner of the window, by use of the ESC-, ENTER- and arrow-keys, or by directly clicking with the mouse into the table. In the latter case, the finger "jumps" to the atomic field in which the mouse cursor is positioned. If the user wants to point to the surrounding (complex) object, the user needs to hold the left button pressed and drag the mouse. The finger will immediately move *Out* to the smallest surrounding object which contains the start and end point of the mouse movement.

Internally, a mouse click gets translated in a sequence of finger operations operating on schema and table trees. Figure 2 below shows the tree representing the table from Figure 1. As can be seen, a finger corresponds to a path inside the tree. The path (node addresses) is stored by means of a stack. Going into an object thus corresponds to a push operation, escaping from an object to the surrounding object becomes a pop operation. These stack operations will show up again in our navigational extension to Tcl in Section 3 below.

Fingers are used internally for all types of operations, e.g. recursively computing tree heights, summing up extensions, sorting objects recursively, drawing the

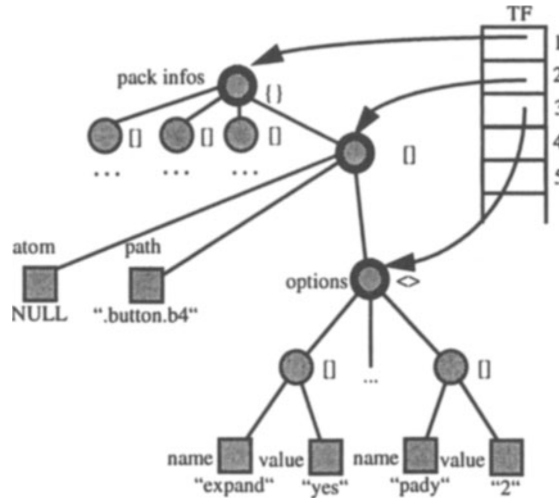


Figure 2 Object tree with stack for a table finger.

schema and the table, "catching" the mouse cursor and quickly redrawing the visible portion, etc.

Atomic and complex objects can be edited using *Insert*, *Delete*, *Backspace*, *Cut-and-Paste* with a clipboard, etc. Note that operations involving the clipboard work with a sub-schema which is established when the first object is moved into it and that objects which are appended to the clipboard must have matching types.

Finally, metadata like the schemes can be inspected (and edited, given proper authorization) as well. For that purpose, a metascheme, called BootScheme is defined within ESCHER. BootScheme is actually a table object under itself, i.e. BootScheme serves as a fixpoint and in fact is circular to handle arbitrary nesting (Wegner, 1989).

4 A NAVIGATIONAL EXTENSION TO Tcl

As mentioned before, we aimed at a declustering of the software which implements the interface. Thus we needed scripting and graphical features as offered by e.g. Tcl or Java. Moreover we wanted to model the extensions to the existing scripting language as closely as possible to the interactive navigational mode which we described above.

Table 1 summarizes these extensions. Clearly, most of them are 1:1-translations of key strokes, menu options, and finger positioning operations through a mouse click or drag..

To understand some of the sample programs, a few additional comments might be helpful. Every newly opened table has a finger *tid*.`root` from which additional fingers can be forked, where *tid* is the Tcl id for the table. When scripts are stored as

Table 1: Basic set of TclDB-commands

Command	Remark
escher	command having to do with the ESCHER database:
boot *path*	get database going, returns 0/1
shutdown	shut database down, returns 0/1
list applications	db is organized into applications, returns list of appl's
list schemas	~ list of schemas (table schemas) defined for an appl.
list tables	~ list of tables defined for an application
application	
select *name*	change to another application, returns application *name*
new *name*	creates a new application, returns *name* of new appl.
delete *name*	removes an application from database, returns *name*
table	command having to do with one or more tables:
open *name*	open a database table given by *name*, returns Tcl table
-id *tid*	identifier *tid*, either system chosen or user defined
list	return *name*1 *tid*1 *name*2 *tid*2 ...
close *tid*	close a table, returns *tid*
new *name scm*	create table giving its new *name* and existing *schema*
delete *name*	delete a table from an application, returns *name*
finger	command having to do with finger (cursor) generation
fork *fid*	create a new finger from existing finger *fid*, returning
-id *newfid*	either system chosen *newfid* or user defined *newfid*
free *fid*	release finger *fid*
list	list all existing fingers, returns *fid1 fid2* ...
-table *tid*	list fingers within a particular table *tid*, returns *fid1* ...
fid **push -first**	with an existing finger *fid* move on first (default)
-last	enclosed object, resp. move finger on last enclosed obj.
-name *attrname*	resp. move into tuple on attribute given by *attrname*
-where *predicate*	move finger on object satisfying *predicate*, returns 0/1
path	move finger along path including attributes and indexes
fid **pop**	move finger *fid* to enclosing object
fid **go**	move finger from one object (tuple, attribute) to
-first	first object within enclosing complex object
-last	last object ...
-back	to previous object (above, to left)
-next	to next object (below, to right)
-forcedup	to previous object (above) within attribute regardless
-forceddown	of tuple boundaries, same to next object (below)
-name *attrname*	within tuple to attribute given by *attrname*
fid **get** [*path*]	return value on which finger rests, optionally extended by path starting with finger position

Table 1: Basic set of TclDB-commands

Command	Remark
fid **type** [*path*] *fid* **attr** [*path*]	return type of object on which finger sits return attribute name of attribute on which finger sits
fid **set** *value* [*path*] **-tonull** [*path*] **-toempty** [*path*]	 set value for atomic object on which finger *fid* sits set complex object to database null turn a set-valued object into an empty set
fid **insert** **-before** [*path*] **-after** [*path*]	inserts a null-object before finger *fid* and move *fid* onto element after finger *fid* and move *fid* onto element
fid **delete** [*path*]	delete object to which finger *fid* currently points and move*fid* to predecessor or to successor or, if neither exists, to enclosing object
fid **on -first** **-last**	return true iff finger *fid* on first element of collection return true iff finger *fid* on last element of collection
fid **isempty** [*path*] *fid* **isnull** [*path*] *fid* **isatomic** [*path*] *fid* **iscomplex** [*path*]	return true iff finger *fid* on an empty collection return true iff finger *fid* on a database null value return true iff finger *fid* on an atomic value return true iff finger *fid* is on a complex value
fid **istype -set** [*path*] **-tuple** [*path*] **-list** [*path*]	return true iff finger *fid* on a set value ... on a tuple value ... on a list value

attribute values, the script comes with a default finger which is forked from the finger which triggered the execution of the script.

Fingers *fid* are treated as commands. Most commands return value, type, or property of the object to which the finger points at the moment. Sometimes, however, it is more convenient to retrieve a value further down without explicitly navigating to it. Therefore, most of the commands, with the exception of **pop** and **on**, take an additional argument *path*, where *path* is an expression built from attribute names, indexes and predicates combined in the usual dot-notation (Subieta et al., 1994).

Similarly, selection predicates using the option **-where** *predicate* can be part of the **go** and **push** methods for fingers and position the finger along the first matching value. This avoids explicit searching for a particular tuple within a collection.

Fingers may also cross table boundaries via object references (links) which are used for object sharing. Note that links are persistent fingers. Higher level procedures can be written which provide relational join features and produce materialized views. One typical generic function is `iterate`, which deals with a common task in nested tables: iterating over a collection. It is defined in Figure 6 used there in a procedure `getOpts`.

5 EXAMPLE: CONNECTING MATHEMATICA TO A DATABASE

To illustrate the principal connection from the object-relational database to the interface via our TclDB scripting facility, consider the following example. It represents a restricted front-end to Mathematica[1]. The data come from two tables. One is called `DEMOPLOTS.tbl`, which is shown in Figure 3 below, and contains argu-

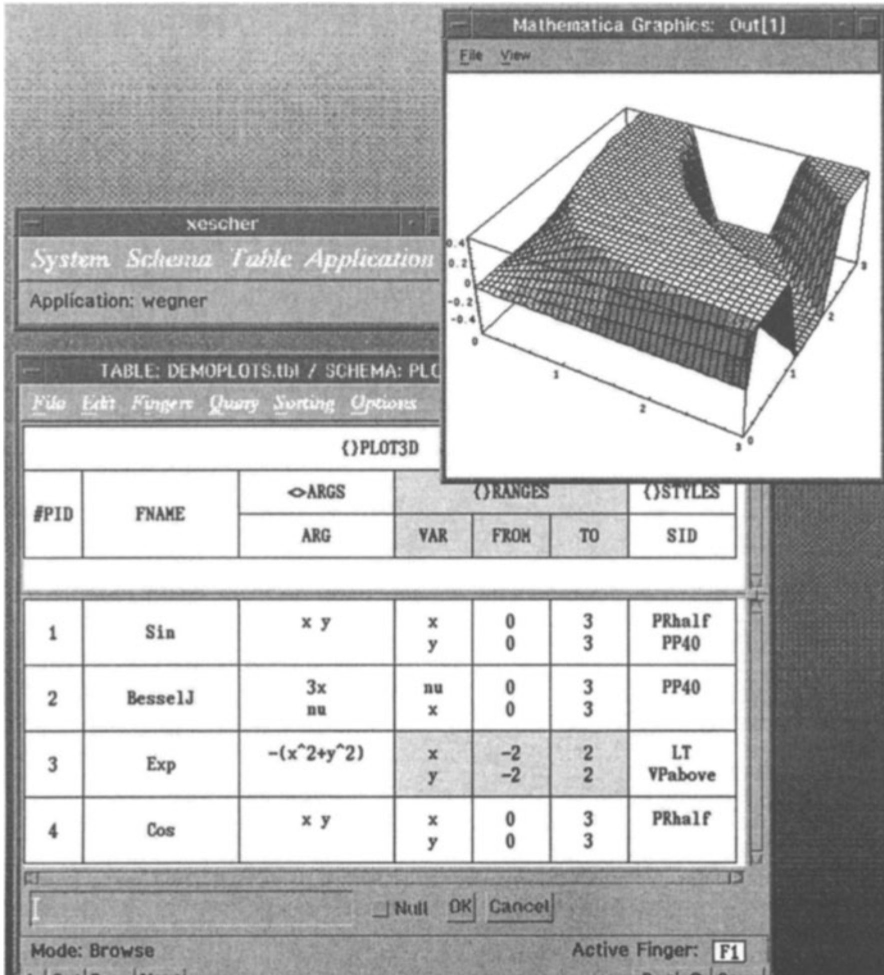


Figure 3 A database tuple piped into Mathematica.

ments and ranges for a set of functions. Any function may be plotted as a 3D graph using style options from the second table `STYLES.tbl` shown in Figure 4. Both

1. Mathematica® is a registered trademark of Wolfram Research, Inc. All examples shown in this section were adapted from (Wolfram, 1993)

TABLE: STYLES3D.tbl / SCHEMA: STYLES.scm / APPL:

File Edit Fingers Query Sorting Options Help

{}STYLES		
SID	OPTION	{}APPLYTO #PID
PRhalf	PlotRange -> {-.5, .5}	1 4
LT	Lighting -> True	3
VPlhcorner	ViewPoint -> {-2, -2, 0}	{}
VPabove	ViewPoint -> {0, 0, 2}	{}
MF	Mesh -> False	{}
SF	Shading -> False	{}
PP40	PlotPoints -> 40	2 1

Null OK Cancel

Mode: Browse Active Finger: F6

In Out Prev Next BeginQ Save

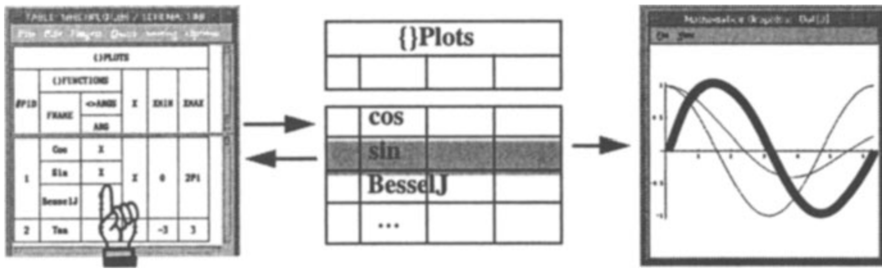
Figure 4 The STYLES table with plot id as foreign key.

tables are connected by a *n:m*-relation which is mapped into a symmetrical nested foreign key attribute.

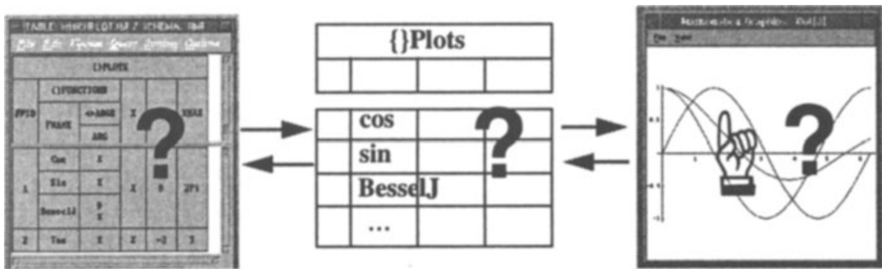
Moving a finger over the table, a user can trigger the execution of a script which collects the data from the tuple on which the interaction finger sits, joins it with the specified styles option and creates a Mathematica input line from it. This is piped into Mathematica for display as shown in the upper right corner of Figure 3. The actual program is only about one page and can be found at our ftp-site.

We should emphasize again that it is not our intention to create a complete front-end to a algebra system like Mathematica. We should also mention that the currently newest version Mathematica 3.0 provides "palettes" which are templates for cut-and-paste input construction. However, they are no general solution to the principal weakness of Mathematica and other algebra systems, namely "the difficulty of getting large lists of data from other programs, such as databases or spreadsheets (Stallmann, 1997)". We may now create multiple copies of a function using cut-and-paste, vary their ranges and styles, view them from different view points, etc. without having to retype the input. We may also query function tables, selecting e.g. all dyadic or triadic functions. We might have additional attributes with comments or joins to their derivatives.

For educational purposes, users might want to create animations and establish



(a) Finger propagation with DB-editor interaction.



(b) Missing inverse mapping with plot interaction.

Figure 5 Open versus closed graphical tools for GUI creation.

collaborative sessions in which lecturer and students navigate inside the generated graphs. If e.g. a diagram shows three superimposed plots as in Figure 5, it is easy for TclDB to sense the presence of another (interaction) finger and then to highlight exactly that plot to which the interaction finger currently points. Similarly, we may hook up a 3D input device to ESCHER which delivers (x, y, z)-coordinates which we store in the styles set of `OPTIONS` as e.g. Mathematica `ViewPoint` values which then creates the desired change in the plot on each change of values.

However, a serious drawback of this piped communication is the impossibility of correlating mouse coordinates on the plot with complex or atomic objects. Because of the closed nature of Mathematica's output it is presently impossible to tell in Figure 5 (b), which of the three functions a user presently clicks on with the mouse and to relate this back to the nested table. Again we should indicate that Mathematica 3.0 now has a function which returns mouse coordinates and if further scaling and positioning information were available, the inverse mapping could be computed.

6 CREATING A GUI FROM A DATABASE

In the previous section we have shown how the ESCHER database editor together with the newly implemented TclDB scripting facility can be connected to an existing graphical tool, in our case to Mathematica. Now we go one step further and take

the interface generation into our own hands.

Until now ESCHER's interface is built upon OSF/Motif with a complex, but fairly standard part concerning menus, windows and buttons and a specific part dealing with clipping, finger movements and visualization for the arbitrarily large NF^2-tables and schemes. Fortunately, ESCHER has a fairly clean layered architecture and all application specific visualization activities as based upon the previously mentioned Object Manager (OM) functions which represents the interface to the ESCHER database engine.

These in turn were mapped into the Tcl-extensions listed in Table 1. The big step was then to design suitable GUI-schemes, create and populate tables for these schemes with the data required to "hold" the interface and finally write generic methods to drive it.

While we cannot go into details for lack of space, one can easily get a general idea by looking at the `pack` procedure in Figure 6. The procedure operates on the table `tcl_pack.tbl` from Figure 1. Its task is to generate Tcl code which packs widgets according to options stored in the just mentioned pack table. Going from back to front, it is easy to see that the last procedure `packWidget` first searches for the proper widget and then calls `getOpts` to extract the options and convert them to a string. It then uses the returned string to initiate packing on the outer level.

Procedure `getOpts` in turn simply iterates over the options list, reads the name and value fields and concatenates their contents to a string in Tcl's format.

The other major generic methods, besides `pack`, are `configure` and `bind`. These and a few others operate on a number of tables, some of which are essential, others optional depending on the type of application. They appear in Figure 7 which sketches the conceptual schema for a general GUI application. One essential table is `Tcl_pack.tbl` which has been shown before in Figure 1, other essential tables are `Tcl_bind.tbl`, `Tcl_widget_options`, `Tcl_pre.tbl` (widget preprocessing), `Tcl_post.tbl` (postprocessing), and `Tcl_widget_`*xyz*`.tbl` which is the necessary widget tree whose shape depends on the *xyz*-application. Other optional tables are `Tcl_menu_entries.tbl`, `Tcl_text_commands.tbl`, and `Tcl_wm.tbl` which governs window manager functions.

Figure 7 also indicates how the tables are related to each other. Note that the `path`-attributes which appears in three of the tables is a foreign key in the widget tree where the path must be concatenated from the `name`-attribute using the usual dot notation.

Obviously, there are several choices for the logical schema design and experience must tell how much normalization is needed. One clear insight is that recursive schemes are needed much like our infinite metaschema because widget hierarchies are unbounded in depth. Stratifying the widget tree into a "flat" table with foreign keys as connectors is certainly not efficient enough and destroys the chance to visually descend into a widget hierarchy.

Another open point is management of widgets and widget-ids at run-time. This relates to the leftmost attribute `#atom` in Figure 1 which presently contains null values (shown as `?d?`). When the actual GUI-interface is created from these data,

```
##############################
# iterate for each element of a complex value
# attention: return in $body doesn't work properly
proc iterate {finger body} {
   if {[$finger iscomplex] != 1} {
      error "iterate on non-complex finger \"$finger\""
   }
   if {[$finger push -first] == 0} {return 0}
   while {1} {
      uplevel 1 $body
      if {[$finger go -next] == 0} break
   }
   $finger pop
   return
}
##############################
# build list of options
proc getOpts {f} {
# $f must sit on a tuple which contains an "options" list,
# which consists of tuples with "name" and "value" attr's.
# Each pair is concatenated, "names" are preceded by a dash.
   set options {}
   $f push -name "options"
   iterate $f {
      lappend options -[$f get "name"][$f get "value"]
   }
   # pop to surrounding tuple
   $f pop
   return $options
}
##############################
# pack widget
proc packWidget {widget} {
   global f_p ;# finger $f_p sits on Tcl_pack's "pack infos"
   # search for $widget's entry, nothing to do if not found
   if {[$f_p push -where {"path == $widget"}] == 0} return
   # get pack options list and pack widget
   globExec pack $widget [getOpts $f_p]
   # pop to "pack infos" set
   $f_p pop
}
```

Figure 6 TclDB script samples for the pack method.

Tcl actually assigns widget ids to instances. Within our development path, database driven GUI management will progress from the design to the run-time stage. Thus, in the future these identifiers will be stored into the tables and serve as keys, have index support and there will be object identifiers allowing fast materialized views.

7 FROM CURSOR TO AGENT

Having a GUI design database which uses the object-relational data model and has a visual editor is already a great help in handling interface creation. In implementations which have higher level descriptive languages like XSQL, OQL, etc. avail-

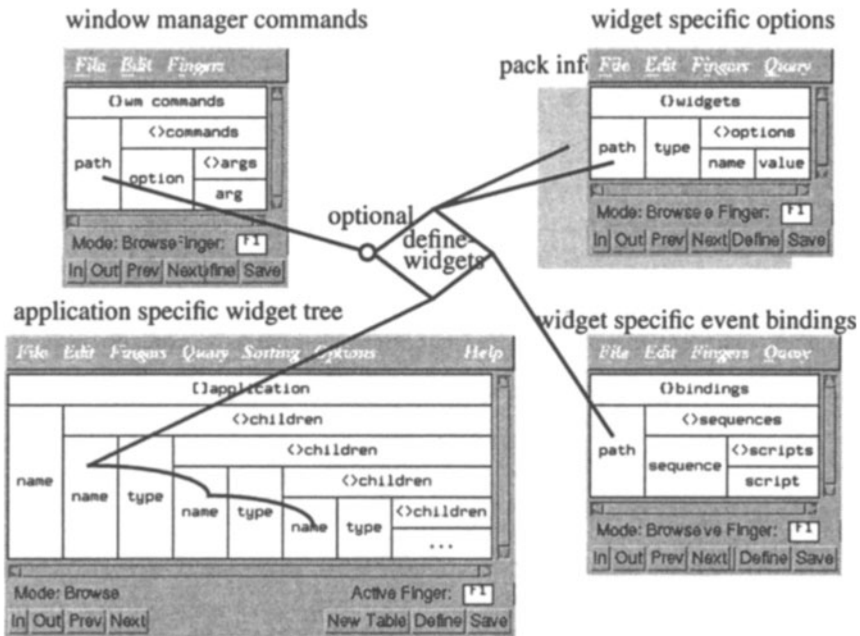


Figure 7 Sketch of GUI conceptual schema.

able, creating a new interface might be supported by formulating the proper queries over the GUI-database. Similarly, it would not be hard to have an interactive GUI-designer map its results to tables instead of libraries.

Ultimately, we plan to have the GUI run from the database as well, i.e. event queues are stored lists of tuples which contain keystroke data, mouse coordinates, etc. Lightweight locks will permit proper synchronization of actions, projections will block out modal attributes, etc. In particular we expect event tracing, one of the biggest problems in GUI-debugging, to become easier by visually stepping through our scripts. Since we have already a way of efficiently visualizing finger positions within nested tables, it is a simple step to visualize the finger which a script receives as parameter (see e.g. `iterate` and `getOpts` in Figure 6).

Fingers will then turn into actors and we can watch "films" with actors fetching events (e.g. a mouse click on a scroll bar), delivering them to the destined widget, creating new internal events for redrawing the slider and scrolling the canvas data. Watching and analyzing these animations will help us write better interfaces and debug them faster.

8 CONCLUSION

We have argued that one way of managing the increasing complexity in GUI development is to move as many interface metadata as possible into ordinary (nested)

tables which are open for inspection and editing. We claim that the nested relational model matches nicely the hierarchical structure of widgets and that resources map easily into attributes.

In general, this proposal corresponds to the GUI database programming strategy reported in (Goyal et al., 1996). However, instead of using logic programming for the "algorithmic side", we suggest declustering of program code by providing smaller chunks of code in form of portable, interpretable TclDB scripts.

As with any interface programming, ease of code generation and performance are the main issues. As for the design stage, we offer a conceptual GUI-schema and generic methods for producing GUI-instances from stored data. These data could be inherited from previous applications, could be modified with cut-and-paste, be materialized views with selections and projections, and could be the output of any of the commercially available interactive GUI-builders.

Performance will also be something to watch, in particular, since we migrate from a very efficient OSF/Motif platform. Speeding up the new interface by introducing visual indexes for graphical queries ("Select all windows which have to be redrawn due to an expose event"), having links for shared objects, pre-computed joins, and materialized views seems a must.

Finally, we feel that our Tcl-extensions provide a seamless integration of ESCHER's navigational paradigm into the scripting language. In fact we could produce scripts interactively by recording key strokes and converting them into TclDB scripts. Furthermore, recent proposals for a cursor-based binding in the SQL/CLI (Venkatro and Pizzo, 1995) point into the same direction.

Many aspects have only been sketched, like event handling and rule mechanisms, when the interface is run from the GUI database. Other needed improvements and further clarifications concern scoping and binding of names (Subieta et al., 1994), visualizing finger movements in trace modes, support for concurrency awareness (Wegner et al., 1996b) and new transactional models (Rusinkiewicz, Klas et al., 1995) in CSCW, support for CAD- and GIS applications, other visualization styles like forms, graphs, table lenses, etc. Now that the scripting facility is up and running we hope to gain more experience with database-driven interfaces.

9 REFERENCES

Brain, M. (1992) Motif Programming - The Essentials ... and More. Digital Press, Burlington, MA.

Buchmann, A.P. (1994) Active Object Systems. in: Advances in Object-Oriented Database Systems (eds. A. Dogac, M. Özsu, A. Biliris, T. Sellis) Nato ASI Series F: Computer and System Sciences, Vol. 130, Berlin, Springer, 201-224.

Date C. and Darwen, H. (1993) A Guide to the SQL Standard. Addison-Wesley, Reading, MA. 3rd edition.

Dayal, S., Buchmann, A.P., and McCarthy, D.R. (1988) Rules are Objects too: A Knowledge Model for an Active, Object-Oriented Database System. Proc. Adv.

in OODBS, Bad Munster, 129-143.

Goyal, N., Hoch, C., Krishnamurthy, R., Meckler, B., and Stone, M. (1996) Is GUI Programming a Database Research Problem? Proc. 1996 ACM SIGMOD Int. Conf., Montreal, Canada, June 4-6, 1996, SIGMOD Record 25:2, 517-528.

Lewis, T. (1996) The Next $10{,}000_2$ Years: Part II. IEEE Computer, 29:5, 78-86.

Ousterhout, J.K. (1994) Tcl and the Tk Toolkit. Addison-Wesley, Reading, Mass.

Rowe, L.R. and Stonebraker, M.R. (1987) The POSTGRES Data Model. - Proceedings of the 13th International Conference VLDB, 83-96.

Rusinkiewicz, M., Klas, W., Tesch, T., Wäsch, J., and Muth, P. (1995) Towards a Cooperative Transaction Model - The Cooperative Activity Model. Proc. 21. VLDB, Zurich, Switzerland, 194-205

Stallmann, F.W. (1997) "Mathematica by example". Comput. Reviews 38:9, 428-9.

Subieta, K.,Beeri, C., Matthes, F., and Schmidt, J.W. (1994) A Stack-Based Approach to Query Languages. in Proc. 2nd Int. East/West Database Workshop (eds. J.Eder and L.A.Kalinichenko), Klagenfurt, Austria, 25-28 Sept. 1994, Springer, London, 159-180.

Sun Microsystems (1995) The Java Language Specification, Vers. 1.0 Beta.

Thamm, J., Thelemann, S., and Wegner, L. (1996) Visual Information Systems - A Database Perspective. Proc. DMS '96 Third Pacific Workshop on Distributed Multimedia Systems, HKUST, June 25 - 28, 1996, eds. David Du and Olivia R. Liu Sheng, Knowledge Systems Institute, Skokie, IL, 274-285.

VAG (1996) VRML Architecture Group: The Virtual Reality Modeling Language Specification - Version 2.0. http://vag.vrml.orh/VRML2.0/FINAL/.

Venkatrao, M. and Pizzo, M. (1995) SQL/CLI - A New Binding Style For SQL. SIGMOD Record 24:4 (Dec.) 71-77.

Wegner, L.M. (1989) ESCHER - Interactive, Visual Handling of Complex Objects in the Extended NF^2 Data Model. Proc. IFIP Work. Conference on Visual Database Systems, Tokyo (April 1989) 277-297

Wegner, L., Thelemann, S., Wilke, S., and Lievaart, R. (1996a) QBE-like Queries and Multimedia Extensions in a Nested Relational DBMS. Proc. Int. Conf. on Visual Information Systems (ed. C. Leung) Melbourne, Australia, 5-6 February 1996, 437-446.

Wegner, L., Paul, M., Thamm, J., and Thelemann, S. (1996b) A Visual Interface for Synchronous Collaboration and Negotiated Transactions. Proc. Advanced Visual Interfaces (AVI'96), Gubbio, Italy, May 27-29, 1996 (eds. T.Catarci, M.F.Costabile, S.Levialdi, G.Santucci) ACM Press, 156-165.

Wilke, D., Wegner, L., and Thamm, J. (1997) Database-driven GUI Programming. Proc. 2. Int. Conf. on Visual Information Systems (Visual'97), San Diego CA (15-17 Dec.) 205-214.

Wolfram, S. (1993) Mathematica - A System for Doing Mathematics by Computer. 2nd ed., Addison-Wesley.