Linking Reactive Software to the X-Window System

Wolfram Schulte Ton Vullinghs

Fakultät für Informatik, Universität Ulm
E-mail: {wolfram,ton}@informatik.uni-ulm.de

November 30, 1994

Abstract

We discuss our experience with linking (existing) reactive applications to X11 based graphical user interfaces. For implementing the user interface we choose to use the Tcl/Tk toolkit, whereas the application itself may be written in any language (even a declarative one) that provides means to perform primitive I/O. The application and the graphical user interface run as separate processes and communicate in either a synchronous or asynchronous way using a bidirectional communication channel. The proposed approach separates software engineering concerns, is easy to use, is reasonably efficient and enables the linking of arbitrary languages to graphical user interfaces.

1 Introduction

Reactive systems are characterized by being event driven, which means that they continuously have to react to external and internal stimuli. Examples include elevators, autonomous robots, operating systems, simulation systems and the user interface of many kinds of ordinary software [6].

This paper outlines a new approach to embed reactive software, supporting a command-line oriented interface, in a graphical user environment. The embedding is done using the commands provided by the X11 toolkit Tcl/Tk [11, 12, 13]. Thanks to the convenient abstraction level of this language, the main problems one has to consider upon creation of the graphical user interface (GUI) are aesthetical ones. The underlying event loops, call back and display routines are all hidden from the programmer.

The command-line interface of the reactive system is linked to Tcl/Tk, using a straightforward method; the application and the GUI run as separate processes and are linked by a pipe. The GUI sends commands to the application, which for its part returns its results to the GUI.

It is obvious, that in the case of synchronous communication, we can easily link any language that contains commands to write and read from a pipe. In addition, we present a method to let the GUI and the application communicate in an asynchronous way, i.e., the number of commands and results that have to be passed between the two processes are not known *a priori*.

The benefits of this approach are manifold:

- *It obeys the principle 'separation of concerns'.* The GUI and the application are each written in languages that are appropriate for their purpose on their particular problem domain with a very precise textual interface defined in between (Section 2).
- *It is easy to use.* Programmers can readily link GUIs to (existing) applications, which may communicate either in a synchronous or asynchronous way, by writing Tcl/Tk code that only has to comply with very simple rules (Section 3).
- *It is not necessary to integrate GUIs in existing languages.* As long as the language provides command-line oriented communication, any language can be combined with Tcl/Tk. For instance, we present a referential transparent solution of a lift control system in the functional language Gofer which is linked to an attractive GUI (Section 4).
- *It is reasonably efficient.* The implementation is sufficiently efficient, as long as only text is communicated between both processes (Section 5).

During the last half year we have encountered a lot of positive experiences with this approach. In summary, we feel that the entire system, combined in the way which is described in detail below, works particularly smoothly as a whole, from the standpoint of both the programmer and the user of the system.

2 Using Tcl and Tk

The combination of Tcl[1] and Tk provides a simple and comfortable programming system for developing simple applications and graphical user interfaces.

2.1 Tcl

Tcl (tool command language) is a small interpretive programming language for controlling and extending applications. It provides variables, procedures, control constructs, and other features. Tcl is an embeddable language, i.e., the language is in fact a library, designed to be linked together with other applications. Tcl only supports the data type string. All commands, expressions, etc. are strings. Depending on the context, strings have to be in some particular format (e.g. numeral strings in arithmetic expressions). Evaluation of strings is done by (recursively) calling the Tcl-interpreter. The example below computes the n-th fibonacci number:

```
proc fib {n} {
  if {$n<=1} {return 1} else {
  return [expr [fib [expr $n - 1]] + [fib [expr $n - 2]]]}}

set e [fib 7]
```

[1] In fact, to have benefit of a more powerful instruction set, we made use of the Tcl extension TclX. Especially the `select` command, needed for asynchronous communication, is a not a Tcl command but is provided by TclX

Using the `set` command we can assign values to strings. Preceding a string by a $-sign means replacing the string by its assigned value. Strings placed between '[' and ']' cause the Tcl interpreter to evaluate the string whereas strings between { and } are passed directly to the interpreter; no substitutions or evaluations are performed.

2.2 Tk

Tk is a toolkit for the X Window System [14] based on Tcl. Items that may appear at the user interface like buttons, labels and menus are called widgets. The Tk toolkit offers a set of widget commands for the creation of user interfaces. All of the functionality of Tk-based applications is available through Tcl, i.e., evaluation of X events in Tk is done by invoking Tcl commands. Since programming in Tcl is rather easy, defining and implementing user interfaces becomes straightforward as well. For the running example in the next section we will use the interface defined by the following program:

```
proc MakeGUI {} {
  pack [button .b -text "Press Me". -width 10  -command Press] -side left
  pack [label  .c -text "Commands:" -width 10] -side left
  pack [label  .d -textvariable com -width  5] -side left
  pack [label  .e -text "Results:"  -width 10] -side left
  pack [label  .f -textvariable res -width  5]}

proc Init {} {
  global com res
  set com 0; set res 0; MakeGUI}

proc Press {} {
  global com res
  incr com; incr res}

Init
```

The procedure `MakeGUI` actually implements the interface, consisting of a button, two text labels and two labels displaying the values of the variables `com` and `res`. Whenever the user presses the button, the procedure `Press` is invoked, thus increasing the value of the global variables `com` and `res` by one. Execution of the program starts with calling the procedure `Init` which initializes the global variables and sets up the user interface.

The `pack` command manages the positioning of widgets in a window. The optional `-side` argument determines the relative position of the next widget to be placed within the available space. Figure 1 shows the state of the widgets after pressing the button 3 times.

3 Interprocess Communication

Although Tcl/Tk is adequate for controlling applications, it is of no use for the development of complex programs, which should be written in languages that are better suited for the particular problem domain. The question arises how to achieve mixed language working. In answering this question we made the following design decisions:

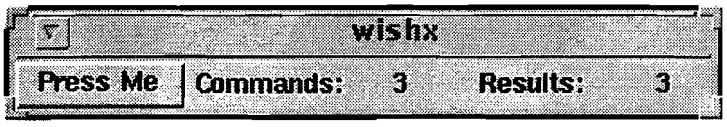

Figure 1: *A small user interface*

- *Changes in existing software should be as small as possible.* Since pipes can easily be established between processes and are the only way to achieve portable interlanguage working, we consider the application and the GUI to be separate processes that communicate over a bidirectional pipe using the command-line interface of the application as the communication protocol.[2]

- *Input and output in either direction should be processed as soon as possible,* e.g., if an event by the GUI or action of the application is generated, it should be sent immediately to the application or GUI, respectively, and should be processed, as soon as it arrives. Sending is achieved by writing commands or results into the pipe, whereas processing means reading the command or result from the pipe and interpreting it.

- *The whole system should be efficient,* or stated in other words, whenever a process has nothing to do, it should be suspended. Since inactive applications have to wait for the next event to process, blocking reads can be used to suspend the application. If the system uses synchronous communication the GUI has to wait until the application sends the next result. To solve this problem, the system can use blocking reads too. However, if the system uses asynchronous communication, the GUI has to wait for a *possible* result of the application. The GUI therefore inspects the pipe's state periodically — during this period the GUI suspends its execution but allows new events to occur.

3.1 Synchronous Communication

To illustrate synchronous communication we return to the previous example, but now we implement it using two processes. The counter `res` is administered in a C application, which writes `res`'s incremented value on standard output whenever it reads an empty line from standard input. Here is the C code of the program `synchronous`:

```
void main(){
  char s[80]; int res = 0;
  while(gets(s)) {printf("%i\n",++res); fflush(stdout);}}
```

Note that we need flushed output in order to process events or actions as soon as possible.

The Tcl program starts, using the Tcl command `open`, with creating the C application `synchronous` as its subprocess. Once it is created, interaction is possible via a pipe, using the commands `puts` and `gets`, i.e., on every press the Tcl program writes a newline into the pipe and reads the C program's result. The procedures `Init` and `Press` are changed accordingly.

[2]Instead of communicating via pipes, it is also possible to make use of the additional software package `expectk` (see [9, 10]). This package extends Tcl/Tk with a number of commands for communication with other applications using virtual terminals. Our approach however, is considerably simpler and makes use of standard Tcl/Tk only.

```
proc Init {} {
  global channel com res
  set com 0; set res 0; set channel [open "|synchronous" r+]; MakeGUI}

proc Press {} {
  global channel com res
  incr com; Write ""; set res [gets $channel]}

proc Write {m} {
  global channel
  puts $channel $m; flush $channel}
```

The resulting system, composed of the C application and the Tcl/Tk process, has the same observable behaviour as the example of Section 2: The number of commands equals the number of results.

3.2 Asynchronous Communication

If an application has to process an unpredictable number of events or generates an unforeseen number of results, we have to deal with asynchronous communication.

We modify the previous example once more. This time the application reads an arbitrary number of times from standard input before it writes an arbitrary number of times the incremented value of **res** on its standard output. The C program **asynchronous** then is defined as follows.

```
void main(){
  char s[80]; int res = 0; int n = 0; int m;
  while(gets(s))
    if(n--==0){
      for(m=rand()%17; m>0; m--){printf("%i\n",++res); fflush(stdout);}
      n=rand()%17;}}
```

The **Init** procedure of the Tcl program is the same as in the previous example except that the C program **asynchronous** is the subprocess to which a pipe is established. On every press of the button a newline is written into the pipe. However, now it is undefined whether the C program replies. Therefore we poll the state of the pipe periodically using the Extended Tcl command **select** which returns the empty string if the pipe is not ready to read. In case the pipe is not empty, we first read the C program's result and then allow the next event to be dealt with using the Tcl command **update**. The modified procedures **Init** and **Press** are listed below:

```
proc Init {} {
  global channel com res
  set com 0; set res 0; set channel [open "|asynchronous"  r+]; MakeGUI}

proc Press {} {
  global com res channel
  incr com; Write "";
```

```
while {"[select $channel {} {} 0]" != ""} {
   set res [gets $channel]
   update}}
```

Even though this example seems to be rather trivial, it demonstrates our strategy towards embedding reactive systems. The user generates events in an unpredictable manner. These events are processed by the read loop of the command-line oriented application. If reading is performed in a blocking manner (as in our example), the application waits for the next event to occur. If non-blocking read is used (which can be realized using the C library function `select`), the application can continue working on some other internal job. In any case, the GUI potentially has to deal with an unknown number of replies. These replies are processed by an iterative non-blocking read, which in each iteration is interruptible to generate new events. Note that there may be several incarnations of the procedure `Press`. However, since the succession of the `select` and `gets` command in one incarnation are not interruptible — new events are only accepted after execution of the `update` command — all incarnations use the same global variables and thus the resulting output is deterministic.

4 The Lift Example

As a more sophisticated example of how to use our approach in combination with a functional language, we present the lift problem. Readers who are not acquainted with the main principles of functional programming may skip this section. For an introduction to functional programming we refer to [3] or [8].

The problem is to simulate the behaviour of a lift in a multi-story building, used by passengers to get to their destination floor.

At each floor we have two buttons (up and down), allowing passengers to call the lift for taking them up or taking them down, respectively. If a button is pressed, it is highlighted to indicate that the request has been recognized. As soon as the request is processed, i.e., the lift has moved to the requested floor and has opened its door, the button's light is turned off. Furthermore, there is a panel in the lift containing floor-number buttons. By pressing one of these buttons, a passenger decides which floor he or she wants to go to. The buttons are highlighted as long as the lift has not reached the corresponding floor. If a lift reaches a floor of destination it automatically opens the door.

In order to close the door, the lift has a close-button. By pressing it, the lift's door is closed again and the lift will start moving if there are any requests from passengers available. If the door is closed, the close-button is disabled.

The lift's behaviour, which corresponds to the lift in our research institute, is based on the following processing strategy:

1. When moving up (down), the lift will process all up (down) or goto requests that can be fulfilled without changing its actual direction.

2. If there is not such an up (down) or goto request, but down (up) requests from floors above (below) the current location exist, then the lift will keep on moving up (down) to fulfill the down (up) request from the highest (lowest) floor.

3. Otherwise, if there are any other pending requests, the lift changes its direction and processes them.

We will now consider the two processes that make up the implementation of the lift system. For the simulation we use an *asynchronous process model.*

The textual interface is defined by the possible events at the user interface and the corresponding actions of the lift:

- The strings denoting the possible *events* are 'close', 'up n', 'down n' and 'goto n', where n denotes the requested floor. Additionally, if the door is closed, the environment generates newlines (clock-ticks) in order to simulate the time interval of a potential move of the lift.
- The output of the lift-controller are the *actions* to move the lift or to open the door. The possible actions are 'Move up', 'Move down', 'Open up' and 'Open down'. The argument of the open action indicates the direction the lift was moving in before opening the door, which is needed to turn off the up or down light indicator, respectively.

4.1 The Controller

The implementation of the lift's operating system is written in Gofer [8], which is a subset of the referential transparent lazy functional language Haskell [7]. Gofer offers several facilities for I/O through which connection to Tcl/Tk is possible. Important however, is to use a Gofer version that uses flushed output.

The lift's state is represented by a four-tuple containing the actual floor number of the lift, a string denoting its direction, the state of the liftdoor, and a list of requests to process. Requests are tagged with a label indicating whether they should be processed while moving up or down.

```
type Floor = Int
type Dir   = String
type Req   = (Int,Dir)
type Reqs  = [Req]
data Door  = Open | Closed
type State = (Floor,Dir,Door,Reqs)
```

The input of the functional program is an infinite sequence of events resulting in a potentially infinite sequence of actions. Since events and actions are represented by strings we can use Gofer's `interact` function and interpret our program as a function `Dialogue`, mapping an input string of characters (from `stdin`) into an output string (on `stdout`).

```
main :: Dialogue
main = interact (running (State 0 Up Open [] []))
```

The function `running` divides the input stream into lines of words. It uses the library function `span`, which partitions a sequence into two lists, in such a way that the first one equals the longest initial segment of the list all of whose elements do not satisfy the given predicate.

```
running :: State -> String -> String
running state input =
  let (headLine, (_:tailLines)) = span (´\n´/=) input
      (newstate, outLine) = event state (words headLine)
  in  outLine ++ (running newstate tailLines)
```

The function **event** computes a new state and returns a possibly empty result string. If the controller accepts a goto-request this request has to be tagged appropriately with 'up' or 'down'. If the request is from a floor below (above) the actual floor, it is inserted as a down (up) request. If the goto request is from the actual floor itself, it is tagged with the lift's actual direction. Requests entered by pressing the up or down button are immediately inserted with the corresponding tag. A close door command changes the door-state to 'Closed'. Finally, if only a newline was sent, the controller will compute the lifts new position.

```
event :: State -> [String] -> (State,String)
event (f,dir,door,reqs) cmd = case cmd of
     ["goto",n]  -> if r > f then ((f,dir,door,(r,"up"):reqs),"")   else
                    if r < f then ((f,dir,door,(r,"down"):reqs),"") else
                                  ((f,dir,door,(r,dir):reqs),"")
                    where r = numval n
     [req,n]     -> ((f,dir,door,(numval n,req):reqs),"")
     ["close"]   -> ((f,dir,Closed,reqs),"")
     []          -> action (f,dir,door,reqs)

numval :: String -> Int
numval cs = foldl (\n x -> 10 * n + ((ord x) - ord '0')) 0 cs
```

Computing the new position is done by the function **action**. The behaviour of this function is in accordance with the strategy presented at the beginning of this section. Whenever the lift has to move up or down the corresponding string is sent; if the lift reaches a floor on which a request has to be processed, the request is removed from the list of requests and the door is opened.

```
action :: State -> (State,String)
action (f,dir,door,[])   = ((f,dir,door,[]),"")
action (f,dir,Open,reqs) = ((f,dir,Open,reqs),"")
action (f,dir,Closed,reqs)
  = ((f,dir,Open,new_reqs),"Open "++dir++"\n")       ,(f,dir) `elem` reqs
  = ((f+one dir,dir,Closed,reqs),"Move "++dir++"\n") ,future_reqs /= []
  = action (f,swap dir,Closed,reqs)                  ,otherwise
  where new_reqs    = filter (/=(f,dir)) reqs
        future_reqs = filter (\x -> (next dir) (fst x) f) reqs
```

The auxiliary functions **swap**, **next** and **one** are given by the following definitions:

```
swap "up"   = "down"        next "up"   = (>)        one "up"   = 1
swap "down" = "up"          next "down" = (<)        one "down" = -1
```

4.2 The User Interface

We define a user interface to visualize the lift problem for a 10-floor building.

The Tk widget **button** is used to simulate a real world button of the lift. Highlighted buttons are represented by disabled Tcl/Tk buttons, whereas all other buttons are active ones. The lift is realized using the scale widget. This widget appears as a linear scale with a

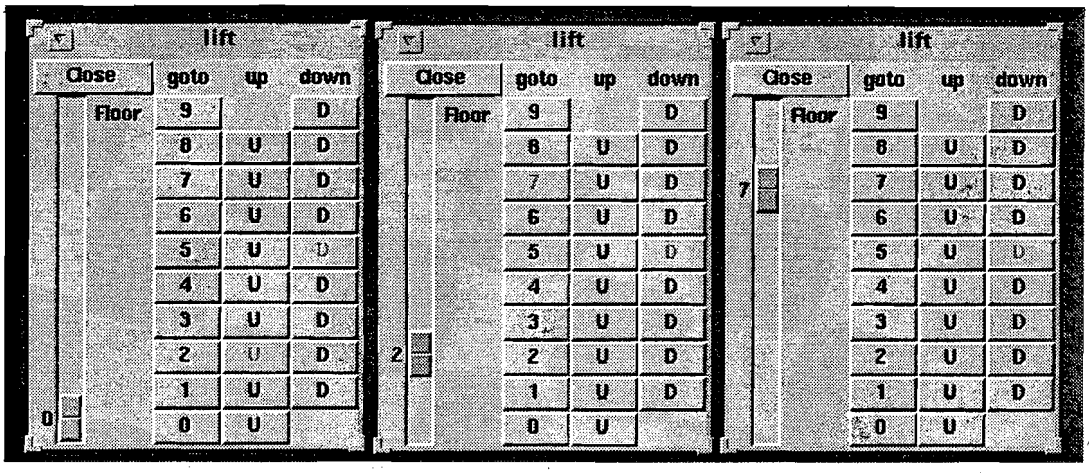

Figure 2: *Three lift states*

slider indicating a value relative to the slider's position. This value corresponds to the actual floor number, the slider represents the lift cabin.

Figure 2 shows three snapshots of the lift system. In the first window we see the lift after receiving requests from floor 2 and 5. The second picture shows the lift after processing the first request. The client who got in decided to go to floor 7. The lift will now first process the up-request at floor 7 and thereafter take care of the down-request at floor 5.

The procedure `Makelift` defines the interface. The procedure as presented below is hand-coded. However, instead of writing your programs by hand it is also possible to use XF [5], a programming environment for interactive construction of Tcl/Tk interfaces.

```
proc Makelift {min max} {
  wm title . lift
  pack [frame .lift] -side left -fill y
  pack [frame .panel]
  pack [frame .panel.names]
  pack [button .lift.door -command Close] -fill x
  pack [scale .lift.cab -from $max -to $min -label Floor] -fill y -expand 1
  foreach e {goto up down} {
    pack [label .panel.names.$e -text $e -width 5] -side left}
  for {set i $min} {$i <= $max} {incr i} {
    pack [frame .panel.floor$i] -fill x -side bottom
    foreach e [list "goto $i" "up U" "down D"] {
      pack [button .panel.floor$i.[lindex $e 0] -text [lindex $e 1] \
           -width 5 -command "Request [lindex $e 0] $i"] -side left}}
  .panel.floor$min.down configure -text "" -relief flat -state disabled
  .panel.floor$max.up   configure -text "" -relief flat -state disabled}
```

Pressing a goto-, up- or down-button will invoke the procedure `Request`, which sends a string to the lift controller. The pressed button remains disabled until the request is processed.

```
proc Request {req floor} {
  Write "$req $floor"
  .panel.floor$floor.$req configure -state disabled}
```

By pressing the close button, the lift is eventually enabled to move again. The close button is not usable until the door is opened by the controller.

```
proc Close {} {
  global busy
  Write "close"
  .lift.door configure -text "Closed" -state disabled
  set busy 1; Getbusy}
```

The procedure `Getbusy` sends newlines (clock ticks) to the controller. Each time a tick is sent, the controller computes the next lift state and possibly returns a result string. As long as no open action is replied by the controller, new clock ticks are generated. The sequence `after 100; update` makes the user-interface wait for 0.1 seconds to make visualisation more realistic and to process new events.

```
proc Getbusy {} {
  global channel busy
  while {$busy} {
    Write ""
    if {"[select $channel {} {} 0]" != ""} {eval [gets $channel]}
    after 100; update}}
```

Result strings are evaluated using the Tcl command `eval`. This command evaluates its arguments as an Tcl script. For example `eval "Open up"` causes Tcl to call the procedure `Open` with argument `up` which enables the close- and floor-number button and reactivates the corresponding up button. Additionally, the global variable `busy` is set to false which terminates the loop in `Getbusy`.

```
proc Open {dir} {
  global busy floor
  .lift.door configure -text "Close" -state normal
  .panel.floor$floor.goto configure -state normal
  .panel.floor$floor.$dir configure -state normal
  set busy 0}
```

Likewise, `Move` is invoked with argument `up` or `down` to move the scale-slider.

```
proc Move {dir} {
  global floor
  if {"$dir" == "up"} then {incr floor} else {incr floor -1}
  .lift.cab set $floor}
```

Finally, the procedure `Init` initializes the communication, sets up the user interface, and opens the liftdoor. The procedure `Write` is defined as before.

```
proc Init {} {
  global channel
  set channel [open {|lift} r+]; Makelift; Open up}

proc Write {m} {
  global channel
  puts $channel $m; flush $channel}

Init
```

5 Discussion

Several case studies have shown the practical value of the proposed method to embed reactive software in Tcl/Tk. The method is for example applied successfully in the design of a GUI for a board game written in Gofer and in the development of a simulator for a stack machine (p-machine). The motivation for these studies was induced by the reimplementation of the user interface of the program transformation system CIP-S [2]. In less than three months we succeeded in rewriting the complete user-interface and linking the existing application.

The most important lessons we learned from these experiences are:

- User interface and application can and should be developed independently of each other. The relation between the two components is very precisely defined. This encourages one to develop the application, extended with an abstract user interface, first, and hereafter the concrete GUI itself.

- Both Tcl/Tk and the method to link applications are straightforward and easy to learn. And since only Tcl/Tk's pure kernel is needed, robustness and stability of the resulting system is guaranteed. In addition, systems become rather portable since Tcl/Tk is available for a wide spectrum of architectures.

- Combining GUIs with arbitrary languages is rather easy. For example, for a lot of functional languages much effort is spent on the integration of graphical I/O (e.g. [1, 4]). In [15] a solution is proposed to link Haskell to Tcl/Tk. The disadvantage of this approach is however that it requires important changes to both the Tcl/Tk and Haskell implementation. By using the standard I/O mechanisms of functional languages in combination with our approach, similar results are obtained in a cheaper way.

- Last but not least, the system is reasonable efficient, although efficiency depends on the kind of communication used. To illustrate this fact, let us return to the example in section 3. If we use synchronous communication over pipes (cf. section 3.1) the total system runs about three times slower than writing the application in Tcl/Tk (cf. section 2.2). If we use asynchronous communication (cf. section 3.2), the situation becomes even more critical: execution times become about ten times longer.

  Although this might seem unacceptable, the following remarks have to be made:

  - For most reactive/interactive applications interaction speed is not critical, i.e., whether we can press a button 5 or 100 times a second makes no difference for the system as a whole, since the latter is no longer realistic.

- The more complex the application, the greater the benefits of the embedding method. For complex applications, most time is spent on the calculations at the application's side, not on communicating. For example it makes no sense to develop an abstract machine code simulator (including an interpreter) in Tcl/Tk.
- In case asynchronous communication decreases system performance too much, mapping on synchronous communication is often a straightforward strategy to improve efficiency.
- And as long as only plain text messages of reasonable size, i.e., less than 250 lines, are passed between the application and the GUI, no significant loss of efficiency arises.

Summarizing, the proposed method meets the ideal requirements stated in section 1 in many respects although there still exist some unanswered questions, for example, how to formally specify the application in combination with its GUI.

References

[1] P. Achten, J. van Groningen, and M. Plasmeijer. High-level specification of i/o in functional languages. In *Glasgow Workshop on Functional Programming.* Springer Verlag, 1992.

[2] F. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Volume II: The Transformation System*, volume 292 of *Lecture Notes in Computer Science.* Springer Verlag, 1987.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall International, 1989.

[4] M. Carlsson and T. Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *Conference on Functional Programming and Computer Architecture.* ACM Press, 1993.

[5] S. Delmas. *XF, Design and Implementation of a Programming Environment for Interactive Construction of Graphical User Interfaces*, 1993.

[6] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems.* Springer Verlag, 1985.

[7] P. Hudak, S. P. Jones, and P. W. (Eds.). Report on the programming language haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[8] M. Jones. *An introduction to Gofer (draft)*, 1993.

[9] D. Libes. expect: Scripts for controlling interactive processes. *Computing Systems*, 4(2), 1991.

[10] D. Libes. X wrappers for non-graphic interactive programs. In *Proceedings of Xhibition 94*, 1994.

[11] J. K. Ousterhout. Tcl: An embeddable command language. In *Proc. USENIX Winter Conference*, 1990.

[12] J. K. Ousterhout. An x11 tooolkit based on the tcl language. In *Proc. USENIX Winter Conference*, 1991.

[13] J. K. Ousterhout. *TCL and TK toolkit.* Addison Wesley, 1994.

[14] R. Scheiffler and J. Getty. The X window system. *ACM Transactions on Graphics*, 5(2), 1986.

[15] D. Sinclair. Graphical user interfaces for haskell. In *Glasgow Workshop on Functional Programming.* Springer Verlag, 1992.

Liste der bisher erschienenen Ulmer Informatik-Berichte

Einige davon sind per FTP von `ftp.informatik.uni-ulm.de` erhältlich
Die mit * markierten Berichte sind vergriffen

List of technical reports published by the University of Ulm

Some of them are available by FTP from `ftp.informatik.uni-ulm.de`
Reports marked with * are out of print

91-01 *Ker-I Ko, P. Orponen, U. Schöning, O. Watanabe*
Instance Complexity

91-02* *K. Gladitz, H. Fassbender, H. Vogler*
Compiler-Based Implementation of Syntax-Directed Functional Programming

91-03 *Alfons Geser*
Relative Termination

91-04* *J. Köbler, U. Schöning, J. Toran*
Graph Isomorphism is low for PP

91-05 *Johannes Köbler, Thomas Thierauf*
Complexity Restricted Advice Functions

91-06 *Uwe Schöning*
Recent Highlights in Structural Complexity Theory

91-07 *F. Green, J. Köbler, J. Toran*
The Power of Middle Bit

91-08* *V.Arvind, Y. Han, L. Hamachandra, J. Köbler, A. Lozano, M. Mundhenk, A. Ogiwara, U. Schöning, R. Silvestri, T. Thierauf*
Reductions for Sets of Low Information Content

92-01* *Vikraman Arvind, Johannes Köbler, Martin Mundhenk*
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets

92-02* *Thomas Noll, Heiko Vogler*
Top-down Parsing with Simulataneous Evaluationof Noncircular Attribute Grammars

92-03 *Fakultät für Informatik*
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen

92-04 *V. Arvind, J. Köbler, M. Mundhenk*
Lowness and the Complexity of Sparse and Tally Descriptions

92-05 *Johannes Köbler*
Locating P/poly Optimally in the Extended Low Hierarchy

92-06 *Armin Kühnemann, Heiko Vogler*
Synthesized and inherited functions -a new computational model for syntax-directed semantics

92-07 *Heinz Fassbender, Heiko Vogler*
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing

92-08 *Uwe Schöning*
On Random Reductions from Sparse Sets to Tally Sets

92-09 *Hermann von Hasseln, Laura Martignon*
Consistency in Stochastic Network

92-10 *Michael Schmitt*
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent Any Linearly Separable Boolean Function

92-11 *Johannes Köbler, Seinosuke Toda*
On the Power of Generalized MOD-Classes

92-12 *V. Arvind, J. Köbler, M. Mundhenk*
Reliable Reductions, High Sets and Low Sets

92-13 *Alfons Geser*
On a monotonic semantic path ordering

92-14* *Joost Engelfriet, Heiko Vogler*
The Translation Power of Top-Down Tree-To-Graph Transducers

93-01 *Alfred Lupper, Konrad Froitzheim*
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications Manager

93-02 *M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch*
The COCOON Object Model

93-03 *Thomas Thierauf, Seinosuke Toda, Osamu Watanabe*
On Sets Bounded Truth-Table Reducible to P-selective Sets

93-04 *Jin-Yi Cai, Frederic Green, Thomas Thierauf*
On the Correlation of Symmetric Functions

93-05 *K.Kuhn, M.Reichert, M. Nathe, T. Beuter, C. Heinlein, P. Dadam*
A Conceptual Approach to an Open Hospital Information System

93-06 *Klaus Gaßner*
Rechnerunterstützung für die konzeptuelle Modellierung

93-07 *Ullrich Keßler, Peter Dadam*
Towards Customizable, Flexible Storage Structures for Complex Objects

94-01 *Michael Schmitt*
On the Complexity of Consistency Problems for Neurons with Binary Weights

94-02 *Armin Kühnemann, Heiko Vogler*
A Pumping Lemma for Output Languages of Attributed Tree Transducers

94-03 *Harry Buhrman, Jim Kadin, Thomas Thierauf*
On Functions Computable with Nonadaptive Queries to NP

94-04 *Heinz Faßbender, Heiko Vogler, Andrea Wedel*
Implementation of a Deterministic Partial E-Unification Algorithm for Macro Tree Transducers

94-05 *V. Arvind, J. Köbler, R. Schuler*
On Helping and Interactive Proof Systems

94-06 *Christian Kalus, Peter Dadam*
Incorporating record subtyping into a relational data model

94-07 *Markus Tresch, Marc H. Scholl*
A Classification of Multi-Database Languages

94-08 *Friedrich von Henke, Harald Rueß*
Arbeitstreffen Typtheorie: Zusammenfassung der Beiträge

94-09 *F.W. von Henke, A. Dold, H. Rueß, D. Schwier, M. Strecker*
Construction and Deduction Methods for the Formal Development of Software

94-10 *Axel Dold*
Formalisierung schematischer Algorithmen

94-11 *Johannes Köbler, Osamu Watanabe*
New Collapse Consequences of NP Having Small Circuits

94-12 *Rainer Schuler*
On Average Polynomial Time

94-13 *Rainer Schuler, Osamu Watanabe*
Towards Average-Case Complexity Analysis of NP Optimization Problems

94-14 *Wolfram Schulte, Ton Vullinghs*
Linking Reactive Software to the X-Window System

94-15 *Alfred Lupper*
Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen