Combining Formal Techniques and Prototyping in User Interface Construction and Verification

Peter Bumbulis* P. S. C. Alencar† D.D. Cowan‡
C.J.P. Lucena§

March 22, 1995

Abstract

In this paper we investigate a component-based approach to combining formal techniques and prototyping for user interface construction in which a single specification is used for constructing both implementations (prototypes) for experimentation and models for formal reasoning. Using a component-based approach not only allows us to construct realistic prototypes, but also allows us to generate a variety of formal models. Rapid prototyping allows the designs to be tested with end users and modified based on their comments and performance, while formal modeling permits the designer to verify mechanically specific requirements imposed on the user interface such as those found in safety- or security-critical applications.

1 Introduction

User interfaces can be difficult and costly to construct; one recent survey estimates that half the development effort for an interactive application is spent on constructing the user interface [MR92]. It is natural to attempt to apply software engineering techniques to reduce this effort. Formal techniques are difficult to apply directly since there is no mathematical characterization of human behavior; even strong proponents of the formal approach to software development

*Peter Bumbulis is a PhD candidate in the Computer Science Department at University of Waterloo. Email: peter@csg.uwaterloo.ca, Fax: (519) 746-5422.

†P. S. C. Alencar is a Visiting Professor in the Computer Science Department at the University of Waterloo, Waterloo, Ontario, Canada and is currently on leave from the Departamento de Ciência da Computação, Universidade de Brasília, Brasília, Brazil. Email: alencar@csg.uwaterloo.ca.

‡D. D. Cowan is a Professor in the Computer Science Department at the University of Waterloo, Waterloo, Ontario, Canada. Email: dcowan@csg.uwaterloo.ca.

§C.J.P. Lucena is a Visiting Professor in the Computer Science Department at the University of Waterloo, Waterloo, Ontario, Canada and is currently on leave from the Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Brazil. Email: lucena@csg.uwaterloo.ca.

have noted that "Formal techniques were not much help to us in designing the user interface." [JJLM91]. Rapid prototyping is usually the methodology of choice for developing user interfaces. Indeed, empirical evidence suggests that "the only reliable method for generating quality user interfaces is to test prototypes with actual end users and modify the design based on the users' comments and performance" [Mye92]. However, the prototyping approach to user interface development is not without drawbacks. One criticism is that it does not provide the same assurance as formal approaches that requirements are being met. This is especially of concern in safety- and security-critical applications.

In part, this has spurred research into drawing together formal specification and rapid prototyping for user interface development [Ale87b]. One common approach is to use a directly executable formal notation to express user interface designs. To take advantage of tools and methodologies, these notations usually are based on an existing (concurrent) formal notation. Statecharts [Mar86], CSP [Ale87a], Petri nets [BP90], temporal logic [Joh91], LOTOS [PF92] and DisCo [Sys94] all have been used. Prototypes are expressed directly as specifications in the formal notation; their behavior is observed by animating the specifications. While there have been various reports of success with this approach [Dix91], there are number of issues which are difficult to address:

1. User interface designers must be fluent in the particular formalism being used. For many formalisms, achieving fluency can involve a significant amount of effort, especially for those not familiar with formal methods.

2. Realistic prototypes (in terms of look-and-feel) are difficult to construct. For example, none of the formal techniques surveyed in [Ale87b], [ABD$^+$89] and [HT90] provide more than rudimentary prototypes for experimentation. As the success of the experimental effort often depends on how realistic the prototype is [RI94], this can be a significant issue. With respect to safety- and security-critical systems this is important since one of the goals of HCI engineering is to reduce the incidence of "user error" [CH94].

3. Formal reasoning is limited to what can easily be expressed in the chosen notation. For example, when using the previously mentioned formal description techniques (FDTs), reasoning is usually limited to behavioral properties.

4. The issue of producing implementations that meet the resulting formal specifications can be difficult to address since, for economic reasons, virtually all user interface software today is implemented using toolkits[1] [Mye94]. These toolkits usually present the developer with a conceptual model that is substantially different from that presented by most user-interface specification languages. For example, while most user-interface specification languages are concurrent, most toolkits are not re-entrant.

[1]We use the term toolkit to refer to tools such as Visual Basic [Mic93] as well as interface libraries such as Motif [Ope91].

5. A formal model must be maintained along with the implementation. This can be hard to justify in situations where the most effective way to develop the next release of an implementation includes rapid prototyping of new functionality within the framework of the existing implementation [Wes93].

In this paper we propose an alternative approach to combining formal techniques and prototyping in user interface construction that addresses these issues. The framework that we propose is component-oriented. It provides the user interface designer with a set of primitive components[2] and a dataflow-based formalism for connecting them: user interfaces are described as directed graphs in which nodes represent components and arcs represent the flow of data between them. The units of data that flow in the arcs are referred to as *events*. From the user interface designer's point of view, events are introduced or *triggered* as a result of a user's actions and then flow from one component to another, being transformed as they go. Components come in two flavors: presentation (menus, buttons, sliders and the like) and application interface (file and database accessors, for example). Each component not only has associated implementation(s) but corresponding model(s) as well.

Rather than basing our framework on a particular toolkit we instead use an interconnection language, IL. As illustrated in Figure 1 (below the dashed line), IL descriptions serve as templates for constructing both implementations

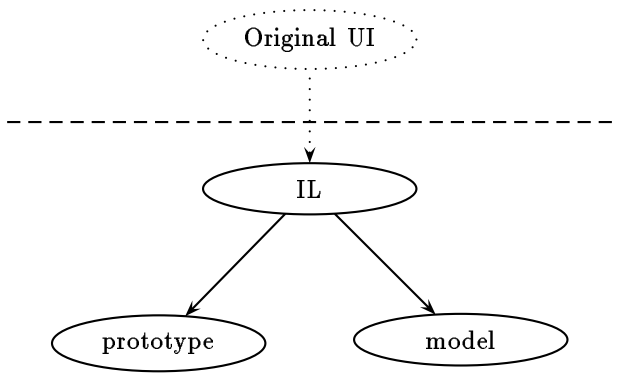

Figure 1: General framework.

for experimentation and models for formal reasoning. Another possible use for

[2]We do not assume the existence of a fixed set of primitive components; this is discussed later.

IL that we are investigating is user interface re-engineering: we can construct an IL prototype of an existing user interface and then use this prototype as a basis for reasoning. This possibility is indicated by the dotted part of Figure 1.

A growing number of commercial UIMSes (PARTS Workbench [Dig92], Visual Age [IBM94] and Visual AppBuilder [Pla94], for example) use a restricted dataflow formalism for specifying user interfaces: restricted in that the topology is (mostly) static, the primitives are objects (widgets) and functions not processes, and the (basic) connectors represent bindings of functions to call sites not FIFO communication channels. We restrict our formalism similarly to ensure that we can realize our prototypes using commonly available toolkits. In particular, it allows us to use any notification-based toolkit[3]. These restrictions also make formal reasoning more tractable.

2 General framework

There are three different roles associated with the development of IL-based user interfaces: the *user interface designer*, or just *designer*, the *developer*, and the *verifier*[4]. The tasks of the designer and developer can be characterized as using and constructing primitive components, respectively. The designer typically requires greater problem domain understanding but less programming skill than the developer. The designer constructs IL descriptions of user interfaces using primitive components supplied by the developer. User interface construction simply consists of selecting and connecting components[5]. If a required component is not available the designer provides the developer with its specification; the developer then uses traditional software development techniques to construct the actual implementation. Designing the primitive components with reuse in mind reduces the amortized cost of development.

The verifier works in concert with the designer and the developer and is responsible for ensuring that prototypes meet formally expressed requirements. Rather than reasoning about implementations directly the verifier generates formal models from the IL descriptions and uses these as the basis for formal reasoning. To have confidence in the results obtained the verifier must ensure that models are accurate, and that reasoning is sound.

The task of ensuring that models accurately reflect implementations can be reduced to ensuring that the primitive components are accurately modeled. If the software development technique used to construct the primitives does not provide the necessary assurance, testing can be used [Stu85, LHM$^+$86]. To ensure that reasoning is sound, some sort of tool assistance is required: practical experience has shown that manual reasoning is less trustworthy than machine-assisted (or machine-checked) reasoning [Coh89, GGH90].

[3]Most commercially available toolkits are notification-based.

[4]There may be many people in each role or one person may perform several roles.

[5]Construction of an interface builder for IL descriptions (such as provided by PARTS Workbench) should be straightforward.

3 The IL formalism

IL, while based loosely on an existing modular interconnection language, Darwin [Imp94], is atypical in that its components are widgets[6] not processes: it is intended for "programming-in-the-small" not "programming-in-the-large".

IL components are either *primitive* or *composite*. Each IL component has a number of *ports* available for binding. Each port has a *polarity*, *requires* or *provides*[7]; only ports of opposite polarity may be bound together.

An IL description of a user interface consists of a set of component descriptions. Each component description consists of a description of the component's interface (i.e. a description of the ports that it makes available for binding) and, for composite components, a description of its implementation. By convention, the user interface described is an instance of the component named `Main`.

We give a taste of IL by example. Figure 2 shows a simple user interface

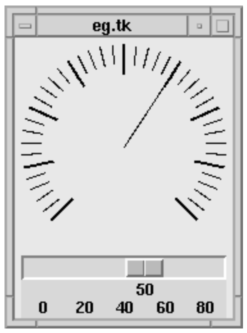

Figure 2: A simple user interface.

consisting of a dial and a slider that track each other. The IL source for this user interface is shown in Figure 3. The first three lines describe the interfaces

```
Frame primitive
Dial changed>int set<int primitive
Slider changed>int set<int primitive

Main {
    f:Frame f.d:Dial f.s:Slider

    f.d.changed --> f.s.set
    f.s.changed --> f.d.set
}
```

Figure 3: An IL description of the user interface in Figure 2.

[6] and functions. We describe only a subset of IL in this paper.

[7] Our terminology is opposite to that used by Darwin (and modular interconnection languages in general): we use the terms provide and require to describe data flow; Darwin uses them to describe services. A port that requires values provides a service.

of the primitive components. Dials and Sliders have set and changed ports that provide (>) and require (<) values[8], respectively. Values sent to a set port update the value of the component; values are issued from a changed port when the component's value changes (either as a result of the user's actions, or as a result of a value being sent to the set port.) Structured names are used to indicate the visual hierarchy: the dial (f.d) and the slider (f.s) are visually contained in the frame (f). Note that (very) simple primitive components have been used to ease discussion.

Instances of our framework are characterized by the primitive components supplied. We are currently investigating a set of primitives which are functionally equivalent to a subset of the primitives provided by PARTS Workbench; while relatively simple, they are useful in practice. We (mechanically) generate Tk/Tcl [Ous94] code for implementations and HOL [GM93] terms for (mechanical) reasoning.

3.1 Constructing Prototypes

To construct a working prototype from an IL description we require for each component a routine to build instances of that component. The developer is responsible for supplying these routines for the primitive components; these routines are automatically generated for composite components. The exact nature of these routines depends on the particular toolkit being used.

Currently we are using Tk/Tcl for prototyping. Tcl (tool command language) is a simple scripting language for controlling and extending applications: the Tcl interpreter is designed to be easily extended with application specific commands. Tk extends Tcl with commands for building Motif-like user interfaces.

Figure 4 contains the code for building Sliders. The code for building Dials is essentially the same except that, as Tk/Tcl does not provide a suitable dial widget, we build one using other widgets. Figure 5 contains the code generated for the IL description in Figure 3. (external.tk contains the code for the primitives.)

Note that as Tcl provides no mechanism for encapsulating data, we do so by giving each component instance a unique name and using this as a prefix for the names of all variables and procedures associated with that instance.

3.2 Constructing Behavioral Models

While the framework introduced in this paper allows for reasoning about various aspects of a user interface, we are most interested in reasoning about their behaviour. For example, we would like to prove that the dial and slider in Figure 2 track each other. To do this we model the behavior of a user interface as a sequence of states with each consecutive pair of states in the sequence representing some action. We model user interfaces as predicates on state sequences:

[8] integers in this case.

```
# Slider`build name:  Create a slider named "name".

proc Slider`build {name} {

    # These should be attributes.

    set V0              0       ;# Minimum value for slider.
    set V1              80      ;# Maximum value for slider.
    set tickInterval    20      ;# Spacing between tick marks.
    set Length          160     ;# Length (in screen units) of slider.

  # Construct the physical representation.

    scale $name -orient horizontal -from $V0 -to $V1 \
        -tickinterval $tickInterval -length $Length -command $name`set
    pack $name

  # Construct the model.

    # Create variable that will hold value of the slider.

    global $name`value
    set $name`value ""

    # If the value of the slider changes $name`set will update the
    # position of the pointer and then invoke $name`changed.

    proc $name`set {value} "
        global $name`value
        if \"!\[cequal \$\{$name`value\} \$value\]\"  \{
            set $name`value \$value
            $name set \$value
            $name`changed \$value
        \}
    "
}
```

Figure 4: Slider in Tk/Tcl.

```
#!/xhbin/wishx -f
source external.tk

proc Main`build {root} {
    Frame`build $root.f
    Dial`build $root.f.d
    Slider`build $root.f.s
    proc $root.f.d`changed {value} "$root.f.s`set \$value"
    proc $root.f.s`changed {value} "$root.f.d`set \$value"
}
Main`build ""
```

Figure 5: Generated Tk/Tcl code for the example.

if P is a predicate representing a user interface $\mathcal{A}$ then $P\,e$ is true if and only if e is a possible behavior of $\mathcal{A}$. If P is a model of Figure 2 then to prove that the slider and the dial track each other we have to prove a theorem of the form $\vdash \forall e.\, P\,e \supset Q\,e$, where Q is a predicate expressing the fact that for all states in a given state sequence, the value of the slider is equal to the value of the dial.

If we are to generate models for IL-based user interfaces mechanically we must be able to model components as predicates and be able to express predicates representing user interfaces in terms of the predicates representing their constituent components. Fortunately such a representation is possible.

We make the observation that the behaviour of a collection of components can be described as a set of mutually recursive functions. For example, if we model the state of the prototype in Figure 2 with the values of two of variables `d` and `s` (representing the values of the dial and the slider, respectively), then its behavior can be modeled with the following ML[9] expression:

```
let rec
    setd v = if (!d = v) then () else d := v; sets v
and actd v = setd v
and sets v = if (!s = v) then () else s := v; setd v
and acts v = sets v
and act = function
      (0,v) -> actd v
    | (1,v) -> acts v
in
    while true do act getaction done;;
```

The function `act` models the possible ways in which the user can interact with the user interface: the user can either set the dial or the slider to some value. The `while` loop models the behavior of the notifier. What follows is a formalization of this observation.

[9]Caml Light [Ler93], not Standard ML.

The formal system we use to express and reason about our models is a version of type theory [Chu40, And86] called *higher-order logic* [Gor86]. Higher-order logic extends first-order logic by allowing higher-order variables (i.e. variables whose values are functions) and higher-order functions (i.e. functions whose arguments and/or results are other functions.) One advantage of using higher-order logic is the existence of reliable and robust proof-assistants such as HOL[GM93] and PVS [ORS92].

The existence of such proof assistants is important for two reasons: 1) experience has shown that machine-assisted proofs are more trustworthy than those done by hand [Coh89], and 2) for some proofs, significant portions can be automated. The proof assistant that we use is HOL. HOL embeds a higher-order logic in the functional programming language ML [CGH$^+$86]. Axioms and primitive rules of inference are encapsulated in an abstract data type `thm`; ML's strong typing ensures that theorems (objects of type `thm`) can only be obtained from these axioms and inference rules. The embedding in ML allows an arbitrary degree of mechanization while still guaranteeing soundness.

We model states as mappings from a variable to values. Rather than expressing behaviors directly in terms of state sequences, we express them in terms of guarded commands [Nel89] and express commands, in turn, as predicates on state sequences using the mechanization of Tredoux [Tre93]. For example, the predicate " := " representing assignment commands is defined as:

$$\vdash_{def}\ \ \forall x\, exp.\ x := exp = (\lambda e.\ \exists s\, s'.\ (e = \mathsf{pair}\,(s, s')) \wedge (s' = \mathsf{bnd}\,(exp\, s)\, x\, s))$$

e is a state sequence representing the assignment of the expression exp to the variable x if and only if for some states s and s', e is the pair (s, s') and s' agrees with s everywhere except possibly on x and $s'\,x = exp\,s$. Note that we model expressions as mappings from states to values. In addition, note that in higher-order logic predicates are Boolean-valued functions.

Components are modeled as predicates on commands. For example we model the Slider of Figure 4 as:

$$\begin{aligned}
\vdash_{def}\ \ & \forall name\ act\ set\ changed.\ \mathsf{Slider}\ name\ act\ set\ changed = \\
& \quad (set = \lambda v. \\
& \qquad\qquad (\lambda s.\ v = s\, name) \longrightarrow \mathsf{skip} \\
& \qquad\qquad [\,](\lambda s.\ v \neq s\, name) \longrightarrow ((name := (\lambda s.\ v));\ changed\, v)) \\
& \quad \wedge(act = \lambda e.\ \exists v.\ set\ v\ e)
\end{aligned}$$

The first conjunct gives the definition of *set* in terms of *changed*, the second conjunct is an expression for the possible effects of a user's interaction with a slider (the user can set it to some arbitrary value v.) Dials are represented similarly.

Given models for its constituent components, we can easily construct a model for a composite component. For example, we model the procedure "`Main`" of Figure 5 as:

$$
\begin{array}{ll}
\vdash_{def} & \forall name\,act.\,\mathsf{Main}\;name\;act = \\
& \exists a_1\,a_2\,a_3\,set_2\,set_3. \\
& \quad \mathsf{Frame}\,(\mathsf{CONS}\,1\,name)\,a_1 \\
& \quad \wedge \mathsf{Dial}\,(\mathsf{CONS}\,2\,name)\,a_2\,set_3\,set_2 \\
& \quad \wedge \mathsf{Slider}\,(\mathsf{CONS}\,3\,name)\,a_3\,set_2\,set_3 \\
& \quad \wedge act = a_1 \;[]\; a_2 \;[]\; a_3
\end{array}
$$

In general, existentially quantified variables are introduced for require ports; their use defines the necessary bindings. Lists of numbers are used for names.

The behavior of the actual user interface is modeled with the following predicate:

$$\lambda e.\,\exists a.\,\mathsf{Main}\;[\,]\;a \wedge \mathsf{do\_od}\,(\mathsf{atomic}\;a)\;e$$

The atomic operator elides intermediate states.

4 Properties

Not only do we have to construct models but we have to formalize properties as well. While formalizing safety and security properties (as well as some generic properties[10]) is relatively straight-forward, formalizing exactly what constitutes a good user interface is an open problem. Many different formalisms and methodologies have been proposed to address this issue; indeed many of the references cited in the introduction can be viewed as approaches to addressing this problem. To take advantage of this body of work one of our goals is to be able to verify that our models possess properties expressed in such formalisms. To this end, we are currently investigating the verification of properties expressed in one of these formalisms: finite state machines [Par69]. One problem that we immediately confront is that most of these formalisms express behaviour as sequence of actions, not states, and further these actions do not directly correspond to our notion of an action[11]. Our solution is to add an extra state variable to our models to record the occurence of these "actions" and to annotate the IL specifications with indications of when they occur.

5 Summary and future work

In this paper we have presented an alternative approach to combining formal techniques and prototyping in user interface construction; one that can take advantage of existing approaches to user interface specification and implementation while addressing the issues raised in the introduction. This approach has been presented in the context of a more general framework: introducing

[10]For example, most of the presentation components that we are working with can be explicitly enabled or disabled. A simple generic property that can be checked is that at all times some component is enabled.

[11]Our actions are simply pairs of states.

the notion of processes (and FIFO communication channels as connectors), for example, would give rise to a much richer formalism.

We have constructed a prototype of the system described in this paper. Currently this consists of IL to Tk/Tcl and IL to HOL translators, and a number of HOL theories and tactics. All of the basic features of the proposed framework have been implemented.

Work is underway on constructing a more complete prototype with a richer set of primitives. Concurrently, we are investigating how to mechanize the various proofs that arise.

References

[ABD+89] Gregory Abowd, Jonathan Bowen, Alan Dix, Michael Harrison, and Roger Took. User interface languages: a survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory Programming Research Group, October 1989.

[Ale87a] Heather Alexander. Formally-based techniques for dialogue design. In *Proceedings of the HCI'87 Conference on People and Computers III*, Systems and Interfaces, pages 201–213, 1987.

[Ale87b] Heather Alexander. *Formally-based tools and techniques for human-computer dialogues.* Ellis Horwood Limited, 1987.

[And86] Peter B. Andrews. *An introduction to mathematical logic and type theory : to truth through proof.* Academic Press, 1986.

[BP90] Remi Bastide and Philippe Palanque. Petri net objects for the design, validation and prototyping of user-driven interfaces. In *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*, Detailed Design: Construction Tools, pages 625–631, 1990.

[CGH+86] G. Cousineau, M. Gordon, G. Huet, R. Milner, L. Paulson, and C. Wadsworth. *The ML Handbook.* INRIA, 1986.

[CH94] Bill Curtis and Bill Hefley. A WIMP no more. *ACM interactions*, pages 23–34, January 1994.

[Chu40] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[Coh89] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–140, June 1989.

[Dig92] Digitalk. *PARTS Workbench User's Guide*, 1992.

[Dix91] Alan Dix. *Formal Methods for Interactive Systems.* Academic Press, 1991.

[GGH90] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging larch shared language specifications. *IEEE Transactions on Software Engineering*, 16(9):1044–1057, September 1990.

[GM93] M. J. C. Gordon and Tom F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, New York, 1993.

[Gor86] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 409–417. North-Holland, 1986.

[HT90] Michael Harrison and Harold Thimbleby, editors. *Formal Methods in Human-Computer Interaction.* Cambridge Series on Human-Computer Interaction. Cambridge University Press, Cambridge, UK, 1990.

[IBM94] IBM. *VisualAge: Concepts & Features*, 1994.

[Imp94] Imperial College of Science, Technology and Medicine. *Darwin Overview*, 1994.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *Mural : a formal development support system.* Springer-Verlag, 1991.

[Joh91] C. W. Johnson. Applying temporal logic to support the specification and prototyping of concurrent multi-user interfaces. In *Proceedings of the HCI'91 Conference on People and Computers VI*, Groupware, pages 145–156, 1991.

[Ler93] Xavier Leroy. *The Caml Light system, release 0.6: Documentation and user's manual.* INRIA, September 1993.

[LHM$^+$86] D. C. Luckham, D. P. Helmbold, S. Meldal, D. L. Bryan, and M. A. Haberler. Task sequencing language for specifying distributed ada systems. In *CRAI Workshop on Software Factories and Ada*, Capri, Italy, May 26–30 1986.

[Mar86] Lynn S. Marshall. *A formal description method for user interfaces.* PhD thesis, University of Manchester, 1986.

[Mic93] Microsoft Corporation. *Microsoft Visual Basic Programmer's Guide*, 1993.

[MR92] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of CHI'92*, pages 195–202, 1992.

[Mye92] Brad A. Myers. *State of the Art in User Interface Software Tools*, chapter 5, pages 110–150. Ablex, Norwood, N.J., 1992.

[Mye94] Brad A. Myers. User interface software tools. Technical Report CMU-CS-94-182, School of Computer Science, Carnegie Mellon University, August 1994.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[Ope91] Open Software Foundation. *OSF/Motif Programmer's Reference, Revision 1.1*, 1991.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, LNAI 607, pages 748–752, Saratoga Springs, New York, USA, June 15–18, 1992. Springer-Verlag.

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

[Par69] David L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 1969 National ACM Conference*, pages 379–385, 1969.

[PF92] F. Paternó and G. Faconti. On the use of LOTOS to describe graphical interaction. In A. Monk, D. Diaper, and M. D. Harrison, editors, *Proceedings of the HCI'92 Conference on People and Computers VII*, pages 155–173. Cambridge University Press, September 1992.

[Pla94] Stephen W. Plain. Novell's visual appbuilder (sidebar to: "radical development"). *PC Magazine*, 13(19), November 1994.

[RI94] James Rudd and Scott Isensee. Twenty-two tips for a happier, healthier prototype. *ACM interactions*, pages 35–40, January 1994.

[Stu85] Howard Sturgis. An effective test strategy. Technical Report CSL-85-8, Xerox Palo Alto Research Center, November 1985.

[Sys94] Kari Systä. Specifying user interfaces in DisCo. *SIGCHI Bulletin*, 26(2):53–58, 1994. Presented at a Workshop on Formal Methods for the Design of Interactive Systems, York, UK, 23rd July 1993.

[Tre93] Gavan Tredoux. Mechanizing nondeterministic programming logics in higher-order logic. Technical report, Laboratory for Formal Aspects of CS, Dept Mathematics, University of Cape Town, Rondebosch 7700, South Africa, March 22 1993.

[Wes93] C. H. West. The challenges facing formal description techniques, October 1993. Invited address at FORTE'93: Sixth international conference on Formal Description Techniques.