ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

C.R. Roast and J.I. Siddiqi, Sheffield Hallam University, UK (Eds)

BCS-FACS Workshop on Formal Aspects of the Human Computer Interface

Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, Sheffield Hallam University, 10-12 September 1996

Using Temporal Logic in the Specification of Reactive and Interactive Systems

R.J. Butterworth and D.J. Cooke

Published in Collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN 3-540-76105-5

Using Temporal Logic in the Specification of Reactive and Interactive Systems

R. J. Butterworth and D. J. Cooke
Computer Studies Department, Loughborough University
Loughborough, Leicestershire, England.

Abstract

Typically formal notations for interactive systems previously presented in the literature (*e.g.* [2, 6, 18]) synthesize two or more languages. We contend that it would be preferable if one were able to use a single soundly based specification language which is expressive enough to capture HCI issues.

Taking a lead from Lamport's Temporal Logic of Actions, (TLA), [14] we outline a language for expressing models of systems based on temporal logic, and make clear the design process we intend this language to be a part of.

We discuss two equivalent specification styles using this language; firstly describing the functionality of the system and secondly describing the interactions of the system. We contend that the second is more 'HCI-centric' than the first.

We discuss other issues raised by the use of the language and set down an agenda for future work.

1 Introduction

The area of HCI and the synthesis of interactive systems is problematic for formal methods. This should not be surprising as HCI is one of the most conceptually complex and information rich areas of computing. It is held that none of the 'all-purpose' formal languages (*e.g.* Z [21] or VDM [13]) have the expressive power to be able to comfortably capture notions crucial to interactive systems. Typically the notations presented previously in the literature (*e.g.* [2, 6, 18]) combine two or more languages in order to gain the expressiveness necessary. This leads to excessive complexity in the notations — the models expressed by such notations may become unwieldy, thus making crucial operations such as refinement more difficult. Furthermore they may impose a 'ceiling' to how abstract the models may be.

It could be argued that the apparent failure of any one formal language to capture HCI related issues demonstrates the inherently intractable nature of much of HCI. Whilst accepting that there is a goodly portion of HCI that would not benefit from being formalised, we take the stance that attempts to capture HCI issues formally highlight the limitations and boundaries of the widely used formal notations rather than the intractable nature of HCI.

1.1 Paper outline

We intend to present a formal language which is to be used in the synthesis of interactive systems.

1.1.1 A design process

We shall first discuss the context of our language; the design process in which we envisage the language being used. (Section 2.) We contend that the classic 'requirements, specification, implementation' design process needs to be augmented and adjusted slightly to accommodate the tricky process of interactive system design.

Our process steers away from the idea of very rigorous interface / functionality separation, instead suggesting that system functionality and the behaviour of the interface should be developed co-dependently.

1.1.2 A formal language

We assert that logical languages (especially temporal action logics) can express state based descriptions and temporal structuring descriptions in a single, holistic language. It has long been known that temporal logics are a good vehicle for expressing system requirements, but recent work [14, 19] has demonstrated that temporal logic can also be used as a specification language.

We have developed a language closely allied to Lamport's TLA (Temporal Logic of Actions) [14]. TLA is one of the more advanced specification languages and we intend to inherit its benefits; notably we can abstract away from processes and processors (in a similar way to the 'DisCo' specification language [11] and its application to interactive systems [23]).

We introduce this language by example in section 3. (The formal syntax and semantics are to be presented in a technical report.) Firstly (section 3.1) by using it to describe very abstract system requirements (the usual domain of temporal logics) and secondly showing how we can state system specifications consistent with these requirements (section 3.2).

We describe two equivalent specification techniques, the first (section 3.2) describes a system in terms of the functionality of its kernel and environment, the interactions between the two being implicit in the specification. The second (section 3.3) describes the possible interactions of the system, the functionality of kernel and environment being implicit in these interactions. Although both techniques are equivalent we suggest that the second is more 'HCI-centric' than the first, as it describes the semantics of interactions. (In this paper we shall concentrate on the semantics of interaction, rather than presentation aspects; we are interested in human-computer interactions rather than the rendering of human-computer interfaces.)

In section 3.4 we look at how we can use the language to describe not only functionality and temporal issues (X happens after Y) but also timing issues (X happens 15 seconds after Y). We also briefly discuss how we can derive measures of usability in terms of performance and error rates (section 3.5) and how we can pass specifications of usability to human factors workers so that they can suggest design strategies which would result in the implementation of more usable systems.

1.1.3 Further work

The design process and formal language are the basis for ongoing work into the formal synthesis of interactive systems. We discuss the direction of this work in section 4.

2 An interactive system design process

We envisage a design process for an interactive system starting at the most abstract level with a statement of requirements for the system with as little as possible implementation bias. From these requirements we derive a specification, which is an abstract model of a system which fulfills these requirements.

A design process for specifying interactive systems is shown in figure 1.

This specification divides the system into two entities; the 'kernel' and its environment, and states a relationship between the two. In an interactive system design process we consider the system kernel to be the automated functionality (*i.e.* the computer) and its environment to be the user population. The specification therefore states the closure of the interface behaviour or *everything* that may happen at the user / computer interface.

To design a 'usable' interface we need to specify what we want to happen at the interface. (*i.e.* what constitutes 'desirable' or 'good' interactions.) This optimum behaviour is a sub-set of the behaviour described in the overall specification (and is similar in concept to 'canonical achievements' in [9]). In this optimal behaviour specification we would describe error-free interactions performed at some optimum interaction pace — the sort of interactions we would expect of an expert user. We can also use this description of an expert user to discuss learnability in terms of how long it takes a novice to attain such optimum interactions. We can then pass this optimal behaviour specification to a human factors expert who will be able to judge what actual interface features, dialogue constructs and such like are going to help the specification to be fulfilled.

Figure 1: An interactive system design process

From these specifications we would proceed to refine and decompose towards an implementation. Such refinements should always be consistent with the overall specification (in that we always produce correct refinements in the usual manner) but should also be biased towards the optimal behaviour specification in that once implemented the system should make it more likely that such optimal behaviour takes place.

The design and implementation of the system functionality (the traditional domain of software engineers) and the designing of the interface (the traditional domain of human factors workers) should therefore proceed co-dependently, both endeavours pushing co-operatively towards a system described by the combination of overall and desired behaviour specifications. This approach to the design process introduces user-centred concepts early, but does not restrict the designer(s) to any one particular interface design or strategy; in classic formal engineering terms we describe *what* we want of the interface without saying *how* this is to be achieved.

3 A scroll bar example

Popular in the literature is the scroll bar. A scroll bar is a graphical representation of the position of a window over some object too large to be fully fitted on the screen. (See figure 2.) A scroll bar is however not purely a passive representation of the window position — certain mouse actions on the scroll bar cause the window position to move.

In this example we start by making it clear that we obviously do not consider a scroll bar to be an end in itself; it is a software sub-system. Let us assume we have come to a point in the design process where it has been decided that a user needs some mechanism for navigating around large data structures and a scroll bar has been decided on for this purpose.

3.1 Requirements for the scroll bar

There are two objects of concern; the scroll bar itself and the windowed data structure it represents. If *SCROLLBAR* is the set of all scroll bars and *WINSTRUC* the set of all windowed data structures then we are going to describe the behaviour of two instantiations of these types; *sbar:SCROLLBAR* and *win:WINSTRUC*.

We assume there is a relationship between scroll bars and windowed structures...

$$rep: WINSTRUC \times SCROLLBAR \rightarrow \mathbb{B}$$

Figure 2: A scroll bar

... which is true iff the scroll bar is a correct representation of the windowed structure. Note that we are not worried here about exactly what a scroll bar is, what a windowed structure is, or what it means for one to be a correct representation of the other. We are abstracting away from all such details.

We may wish to state that the relationship *rep* always holds between *sbar* and *win*. We would state this formally as follows...

$$\text{strictReq} \triangleq \Box(\text{rep}(\text{win}, \text{sbar})) \quad (1)$$

The temporal operator $\Box$ reads ‘always’ (or ‘henceforth’). Hence formula 1 reads ‘it is always the case that *sbar* is a correct representation of *win*.’ This is an unreasonable requirement however; it requires that the scroll bar and windowed structure are *always* related by *rep*; once one entity changes the other must be updated simultaneously, which we cannot implement without the mythological infinitely fast machine.

A more reasonable requirement is that if ever the scroll bar and windowed structure are not related by *rep* then they must become so in the future.

$$\text{req}_1 \triangleq \Box(\neg \text{rep}(\text{win}, \text{sbar}) \Rightarrow \Diamond \text{rep}(\text{win}, \text{sbar})) \quad (2)$$

The temporal operator $\Diamond$ reads ‘eventually’. Hence the above formula reads ‘it is always the case that if the scroll bar is not a correct representation of the windowed structure then it must become so eventually.’

However, formula 2 is not a complete requirement for the scroll bar. Imagine a situation where the scroll bar button is moved; what we want is the window to move correspondingly. The requirement does not guarantee this; it allows a situation where the user moves the scroll bar button, but instead of moving the window the scroll bar may simply move back to its original position. Even worse, it allows for situations where both the scroll bar button and window move arbitrarily as long as they finish up in a position where *rep*(*win*, *sbar*) holds.

Formula 2 is effectively (though not exactly) a safety requirement [3]. We need to add (what is effectively) a liveness requirement stating that a change to the scroll bar results in a change in the windowed structure and *vice versa*.

Using the temporal operator $\leadsto$ (which reads ‘then’ or ‘followed by’) we can describe formally *sbar* changing its value; $\text{sbar} = x \leadsto (\text{sbar} = y \wedge x \neq y)$ ¹. Once *sbar* changes then we need to ensure that there is some future point where *win* has (possibly) changed to accommodate, but *sbar* remains unchanged. Note that *win* need not have to change; changing the value of *sbar* does not automatically mean that *rep*(*win*, *sbar*) no longer holds.

¹The quantities *x* and *y* are existentially quantified, but this is omitted so as not to clutter the formula.

The formula req_2 puts all this together and states the same thing for changes in win .

$$req_2 \triangleq \square \left(\begin{array}{c} sbar = x \rightsquigarrow (sbar = y \wedge x \neq y) \Rightarrow \Diamond(sbar = y \wedge rep(win, sbar)) \\ \wedge \\ win = x \rightsquigarrow (win = y \wedge x \neq y) \Rightarrow \Diamond(win = y \wedge rep(win, sbar)) \end{array} \right) \quad (3)$$

The requirement for how the window position and scroll bar interact is the conjunction of these ‘safety’ and ‘live-ness’ requirements.

$$req \triangleq req_1 \wedge req_2 \quad (4)$$

3.2 System specification

In the above section we have shown how temporal logic can be used in the ‘traditional’ way to state requirements for a system. We now show how we can use a temporal action logic to express the system specification; usually the domain of state based languages such as VDM [13] and process algebras such as CSP [10].

A specification states some initial condition for the system and describes the actions that can henceforth occur in the system. Actions have duration and express the transformation of a system from one state to another. We express them by using undecorated variable names to describe variables in the start state and variables decorated with a dash (') to describe variables in the end state of the action. For example the action of incrementing the variable x is given by $x' = x + 1$.

In a very abstract way there are two actions the user can perform and two actions the system can perform; the user can alter the scroll bar or he can alter the window position. We can state these actions very simply. We also include boolean variables ($barAltered:\mathbb{B}$ and $windowAltered:\mathbb{B}$) which are set to true once the user has moved either the scroll bar or window respectively.

$$alterBar \triangleq \text{enable} : \text{true} \\ sbar \neq sbar' \wedge \\ barAltered' \quad (5)$$

$$alterWindow \triangleq \text{enable} : \text{true} \\ win \neq win' \wedge \\ windowAltered' \quad (6)$$

An action includes an enabling predicate which we keep syntactically separate for clarity. To keep things simple we assume that the user can move either the scroll bar or window position at any time (in more complicated examples this may not be the case of course) hence the enabling predicate is true and would usually be omitted.

The user can perform either of these actions, so overall the user action can be described as the disjunction of these two actions.

$$user \triangleq alterBar \vee alterWindow \quad (7)$$

There are two system actions which react to these user actions, namely the actions of updating the window position so that it is correct with respect to a new scroll bar position or *vice versa*.

$$updateBar \triangleq \text{enable} : windowAltered \\ rep(win, sbar') \wedge \neg windowAltered' \quad (8)$$

$$updateWindow \triangleq \text{enable} : barAltered \\ rep(win', sbar) \wedge \neg barAltered' \quad (9)$$

The overall kernel action is the disjunction of the two kernel actions.

$$kernel \triangleq updateBar \vee updateWindow \quad (10)$$

So we have described, in a quite abstract manner, the functionality of the system. As asserted in [16] however, this is not sufficient to define the *behaviour* of a system — we need to describe under what circumstances actions occur. Obviously an action can only occur when its enabling condition is fulfilled, but does the fulfillment of such a condition mean that an action *may* occur or it *must* occur? We need to distinguish between the two — *alterBar* is always enabled, but this does not mean that a user must always be moving the scroll bar.

We take from deontic logic the concepts of permission and obligation. We require that an action requested of the kernel is eventually undertaken (when adequate processing resources are available). On the other hand there is no compulsion on the users to actually move the scroll bar or window position. Hence we use special notation; angle brackets $\langle \dots \rangle$ ² round an action to indicate permission and square brackets $[\dots]$ to indicate obligation. Hence $\langle A \rangle$ is read as ‘if the action *A* is enabled it may occur’ and $[A]$ is read as ‘if action *A* is enabled it must occur.’

As well as the actions we need an initial condition; namely that $rep(win, sbar)$ holds and that the flags are false.

$$specInit \triangleq rep(win, sbar) \wedge \neg barAltered \wedge \neg windowAltered \quad (11)$$

So the specification for the system states the initial condition (terms in temporal logic formulae not guarded by a temporal operator are said to hold at ‘time zero’) and that henceforth the user actions may happen and the kernel actions must happen whenever they are enabled.

$$spec \triangleq specInit \wedge \Box[kernel] \wedge \Box\langle user \rangle \quad (12)$$

Formula 12 is the typical form of a system specification; some characterisation of the initial state, some action(s) that the environment is permitted to perform and some action(s) that the kernel must perform when they become enabled. In line with TLA we could also conjoin fairness conditions into the specifications (but they are not necessary in this example).

3.3 Specification by reactions

It is obvious from the the above example that the user action *alterBar* causes the kernel action *updateWindow* because the result of the user action implies the enabling condition of the kernel action and the obligatory nature of the kernel action assures it must happen once enabled.

We call such causal groupings of user and kernel actions ‘reactions’ and are typified by an ‘invocation’ (the user action(s)) and a ‘response’ (the kernel action(s)).

If the $\circ$ operator defines the sequential composition of two actions then we may rewrite the specification (formula 12) as follows...

$$reactSpec \triangleq init \wedge \Box\langle alterBar \circ moveWindow \vee alterWindow \circ moveBar \rangle \quad (13)$$

...which states that it is always possible for the reaction *alterBar* $\circ$ *moveWindow* or *alterWindow* $\circ$ *moveBar* to occur.

We consider a reaction to be a unit of interaction and we would specify interactions as temporal structurings of reactions. Such structurings may become rather involved and complicated and a single thread model of interaction is possibly inadequate. We need to use partial ordering techniques similar to those suggested in [7] in order to clearly express interactions.

Thinking in terms of what interactions a system undertakes is possibly a more ‘HCI-centric’ view of system specification than the technique described in section 3.2, but the two are obviously interrelated.

²Note that our use of brackets is not at all the same as Lamport’s in TLA

3.4 Timing constraints

One of the problems with using formal methods in HCI is that formal notations tend to abstract away from the notion of explicit time which can be crucial to expressing usability (see [12]). Like Pnueli's TLR [19], our specification language is based on a real time index (unlike TLA, however see [1]) so that we can discuss such issues.

In the above example we have carefully side-stepped the issue of timing with the result that a great many obviously useless systems could be implemented that are consistent with the above specification. There are no constraints on how quickly we wish the kernel to reinstate the $rep(win, sbar)$ relationship once the user has altered the scroll bar or data structure.

We introduce a new eventually operator which is parametrised by an amount of time. Hence $\Diamond_t \phi$ reads ' ϕ becomes true within time t ' and we can rewrite formula 2 to make use of this.

$$req_1 \triangleq \Box(\neg rep(win, sbar) \Rightarrow \Diamond_\delta rep(win, sbar)) \quad (14)$$

δ is some amount of time considered to be unnoticeably quick. (Around 150ms from the heuristics in [20].)

Now we have changed the requirements we also need to change the specification to reflect this. We use the special variable t to represent time. We assume there is an implicit clock action that always advances t (in a regular manner thereby avoiding philosophical problems with 'Zeno's paradox'). This clock action is the only action that can advance t hence all other actions treat t as a 'read-only' variable.

$$\begin{aligned} updateBar &\triangleq \text{enable} : windowAltered \\ &\quad rep(win, sbar') \wedge \neg windowAltered' \wedge \\ &\quad t' \leq t + \delta \end{aligned} \quad (15)$$

$$\begin{aligned} updateWindow &\triangleq \text{enable} : barAltered \\ &\quad rep(win', sbar) \wedge \neg barAltered' \\ &\quad t' \leq t + \delta \end{aligned} \quad (16)$$

We may hit the problem here that δ is simply too fast to implement; a common worry when specifying explicit timing requirements. We have to be aware that we can specify unimplementable systems, so there may be more cycling in the development process than would normally be expected. There are other strategies for overcoming such problems, (see [5]) for example we could be very liberal when we decide what it means for $rep(win, sbar)$ to hold.

3.5 Desired interactions

Having specified what the scroll bar system does we now need to think about exactly what we want the users to do with it. A scroll bar is a tool for enabling the users to navigate through the windowed structure. We require of the scroll bar that it allows for both quick and accurate navigations.

3.5.1 Quick navigations

What do we mean by 'quick'? It is a rather subjective term, yet we can apply a more objective measuring scheme in terms of user performance. Given the task of navigating from point A to point B in a structure we can objectively measure the speed of the user accomplishing this task, both with reference to time taken and number of invocations necessary. Norman [17] states that user satisfaction is a trade off between the two.

We can therefore state a specification of desired interactions with reference to user performance. This specification should then be passed to a human factors expert whose task it is to judge which 'direction' of design should be pursued in order that users are more likely to interact with the computer according to the desired interaction specification. We envisage that to promote quickness the human factors expert would prescribe the need for some operations which can move the window a considerable distance (such as page up or down operations, or the ability to drag the scroll button).

3.5.2 Accurate navigation

Again ‘accurate’ is worryingly subjective but we believe that we can measure such things in terms of error rates; if we assume once more that the user is navigating from point A to B, how often does the user miss point B and when he does, how much does he miss it by? We can specify the accuracy of navigation in terms of maximum tolerable error rates.

Of course it would be fallacious to assume there is a clear separation between speed and accuracy; accuracy will greatly effect the speed. Another important factor is the clarity of the feedback provided by the scroll bar (*i.e.* how useful as a representation of the position of the window it is) as this will have a large impact on accuracy.

Accuracy requires the provision of commands that move the window atomically (*i.e.* the smallest movement possible) and that the large moves such as page up move the window predictably.

We can also see from this example that there is no definite separating line where software engineering stops and human factors expertise starts. In other words we do not separate core functionality and interface (see arguments in [4, 22]). We have shown the human factors expert having a pivotal role in decisions early on in the specification process — judging what navigation commands should be available to the user (page up, page down, *etc.*) as well as designing mechanisms for presenting those commands usefully to the user.

4 Conclusions and further work

Because our language is closely allied to TLA we are confident that we can inherit its theoretical basis and this will prevent us from having to re-invent formal wheels.

4.1 Is our language a specialisation or extension to TLA?

We have, however, introduced deontic concepts not found in TLA. We need to be sure that these concepts specialise TLA rather than extend it. To ensure ourselves of this, let us take a close look at what constitutes a specification in TLA and our language.

4.1.1 What is in a TLA specification?

A TLA specification states that some initial predicate holds and henceforth some action occurs which may stutter but is live. In all but the most trivial systems this action is a disjunction of several sub-actions. There is typically a fairness condition placed on these sub-actions.

4.1.2 What is in our specifications?

Our specification also states that some initial predicate holds and henceforth some action occurs. However our action is built of a more structured disjunction of actions. At any one time there will be a set of actions that are enabled, some of which may be obligatory and some of which may be merely possible. (We should always include a null action, which is always possible, so at any one time there is at least one possible action, even if that action is ‘do nothing’ and hence the system cannot lock.)

Typically actions that are merely possible are not live (in that we cannot guarantee they will occur) whereas obligated actions are live — once enabled they must occur. In terms of a computational model of our specification language obligated actions have a higher priority than possible actions. Presented with a set of enabled actions a system should perform the actions that are obligatory by preference and the possible actions (including the null action) secondarily.

4.1.3 What is the difference?

We can therefore discuss our notions of permission and obligation (which, though derived from, are *not* the same as those given in [16]) in terms of liveness of sub-actions. Alternatively we could think of the deontic operators in terms of fairness — obliged actions are not fair, they must occur exactly the same number of times as they are enabled.

TLA has the apparatus to deal with discussing liveness and fairness and hence our language *is* a specialisation of TLA.

4.2 Complexity

The example in this paper dealt with a small scale example in a very abstract way. We need to feel confident that we can introduce complexity in a ‘real-life’ situation. Complexity stems from either more reactions or more complex reactions. Hence in complicated specifications the structure of the specification should not change, only what is in that structure.

4.3 Further work

One of the objectives of this work is to provide syntactic sugaring so that we can specify systems in terms of reactions and interactions built from them in a way comprehensible to HCI workers with little interest in formal methods. In particular we are keen to investigate the use of graphical notations, using such work as Harel’s statecharts [8] and TLA in pictures [15] as a starting point.

In this paper we have only glimpsed what we are likely to want to express of optimal interaction specifications, rather than how we are actually going to do it. We could think of an optimal interaction being one that contains only ‘good’ reactions (and no unproductive interaction loops). Once invoked a reaction should always produce a response, but what if the operation the user is trying to invoke cannot occur? For example the user invokes page down when the window is already at the bottom of the data structure. We could prevent such invocations by disabling them (perhaps greying out the page down button) or we could allow the invocation but respond with some error signal. A ‘good’ reaction is one that does what the user expects of it.

We have discussed some rather subjective terms (goodness of a reaction, error rates, user performance *etc.*) which make very little sense if discussed in the language of discrete mathematics. We intend to further advance our specification language by the introduction of apparatus that can deal with stochastic and approximate models. We believe this will help push our formal work further into the realms of experimental psychology and the ‘approximate science’ advocated by Norman [17].

References

- [1] Abadi, M. and Lamport, L. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.
- [2] Abowd, G. *Formal Aspects of Human-Computer Interaction*. PhD thesis, University of Oxford, 1991.
- [3] Alpern, B. and Schneider, F. B. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
- [4] Cockton, G. Where do we draw the line? Derivation and evaluation of user interface software separation rules. In Harrison, M. and Monk, A. F. (Eds), *People and Computers: Designing for Usability*, pages 417–432. Cambridge University Press, 1986.
- [5] Dix, A. J. The myth of the infinitely fast machine. In Diaper, D. and Winder, R. (Eds), *People and Computers III, HCI’87*, pages 215–228. Cambridge University Press, 1987.
- [6] Duke, D. J. and Harrison, M. D. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–36, 1993.
- [7] Duke, D. J. and Harrison, M. D. Event model of human-system interaction. *Software Engineering Journal*, 10(1):3–12, 1995.
- [8] Harel, D. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

- [9] Harrison, M. D., Blandford, A. E., and Barnard, P. J. The requirements engineering of user freedom. In Paternó, F. (ed), *Proceedings of the Eurographics Workshop on Design Specification and Verification of Interactive Systems*, pages 181–194. Eurographics, 1995.
- [10] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall International, 1985.
- [11] Jarvinen, H. M. The design of a specification language for reactive systems. PhD thesis, Tampere University of Technology, 1992.
- [12] Johnson, C. Applying temporal logic to support the specification and prototyping of concurrent multi-user interfaces. In Diaper, D. and Hammond, N. (Eds), *People and Computers VI. Proceedings of the HCI91 Conference*, pages 145–156. Association of computing machinery, Cambridge University Press, 1991.
- [13] Jones, C. B. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice Hall International, 1986.
- [14] Lamport, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
- [15] Lamport, L. TLA in pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, 1995.
- [16] Maibaum, T. Temporal reasoning over deontic specifications. In *Deontic Logic in Computer Science — Normative system specification*, pages 141–202. Wiley, 1993.
- [17] Norman, D. A. Cognitive engineering. In *User Centered System Design – New Perspectives on Human-Computer Interaction*, pages 31–61. Lawrence Erlbaum Associates, 1986.
- [18] Palanque, P. A. and Bastide, R. Petri net based design of user-driven interfaces using the interactive cooperative objects formalism. In Paternó, F. (ed), *Proceedings of the Eurographics Workshop on Design Specification and Verification of Interactive Systems*, pages 215–228. Eurographics, 1995.
- [19] Pnueli, A. System specification and refinement in temporal logic. *Lecture Notes in Computer Science*, Vol 652, pages 1–38, 1992.
- [20] Shneiderman, B. *Designing the User Interface*. Addison-Wesley, 1987.
- [21] Spivey, J. M. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice Hall International, 1989.
- [22] Storrs, G. A conceptual model of human-computer interaction. *Behaviour and information technology*, 8(5):323–334, 1989.
- [23] Systá, K. Specifying user interfaces in disco. *SIGCHI Bulletin*, 26(2):53–58, 1994.