Declarative Graphics And Dynamic Interaction

C.W. Johnson and M.D. Harrison

The Human Computer Interaction Group, The Dept. of Computer Science,
The University of York, Heslington, The United Kingdom, YO1 5DD.
E-mail: johnson@minster.york.ac.uk

Abstract

First order logic provides a means of integrating the specification and prototyping of interactive systems. It can describe graphical images in a declarative and order independent manner. It supports the definition of abstract devices which avoid the complexity of representing 'raw' input from a variety of physical devices. The following pages show how such techniques must be extended in order to prototype and reason about dynamic interaction with graphical interfaces. The incorporation of a temporal ordering into logical specifications provides a means of describing changes in the structure of graphical images. It can also identify the sequencing which may be implicit within specifications of interactive dialogues. This paper describes how PRELOG, a tool for Presenting and REndering LOGic specifications of interactive systems, has been extended to include a temporal logic interpreter.

Keywords: Declarative graphics, temporal logic, specification, prototyping.

1. INTRODUCTION

Declarative approaches to the specification and implementation of graphical images offer a number of benefits for the design of interactive systems. For example, constraint based programming languages provide a means of expressing consistency requirements between the image and behaviour of an interface [22]. Previous work, by the authors, has argued that the declarative powers of first order logic makes it ideally suited for both the specification and prototyping of interfaces [10]. Executable subsets of the formalism, such as that supported by PROLOG, provide a means of rapidly deriving prototypes from abstract specifications. There is a close correspondence between specifications written in first order logic and programs which satisfy those specifications implemented in PROLOG [12, 5]. It can be used to avoid the introduction of unstructured bitmaps into a formal specification. It supports the incremental modification of images and avoids the order dependence of procedural languages [17]. First order logic supports the description of device abstractions which avoid the complexity of representing 'raw' input from

a variety of physical devices [8]. Abstract requirements may be specified for an interface without necessarily considering low level implementation details. There are, however, a number of problems which limit the application of this formalism to the design of interactive systems. For instance, the absence of state information may lead to a reliance on implementation strategies and side-effects which are not explicit within a specification.

2. FIRST ORDER LOGIC AND DECLARATIVE GRAPHICS

Complex graphical images can be described in terms of a number of component parts. These may, in turn, be described in terms of their parts. Primitive objects, such as lines or regions, support the graphical structure [10]. These part-whole hierarchies

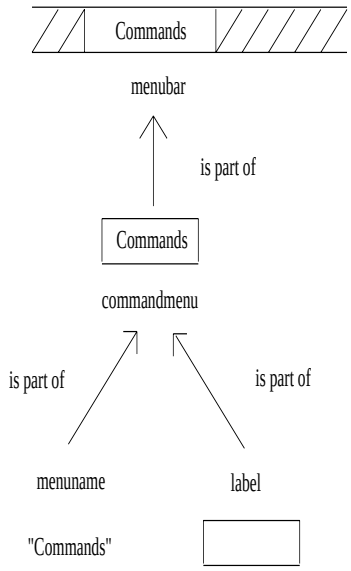

Figure 1: The part-whole hierarchy of menubar.

may be specified using first order logic [8]. For example, a label might be part of a pull down menu which is, in turn, part of a menu bar:

$part(menubar, commandmenu)$.
$part(commandmenu, label)$.
$part(commandmenu, menuname)$.
$graphics(label, line(0.0, 0.0, 0.3, 0.3))$.
$text(menuname, 'Commands'))$.

Figure 1 illustrates the resulting graphical structure. Part-whole hierarchies offer a number of benefits to the interface designer [2]. They are device and order independent. The image is not determined by an execution sequence. The appearance of the interface may be described without reference to particular devices.

Graphical structures, like that presented above, are not easily adapted to describe the dynamics of interaction. For example, a user might select *label* in order to pull down the menu. This would reveal a new structure including command options:

$part(commandmenu, monitoritem).$
$part(commandmenu, restartitem).$
$part(commandmenu, manualitem).$
$part(commandmenu, offitem).$

This is illustrated by figure 2. In order to describe this change in appearance the designer

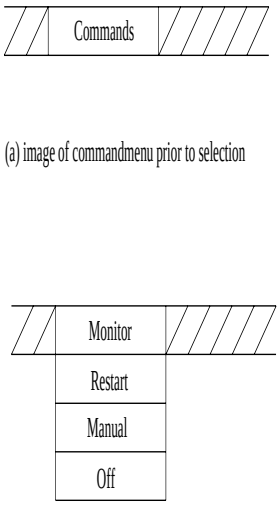

Figure 2: The image of *commandmenu* before and after a selection event.

must maintain a distinction between the structure of the image before and after selection. For example, the following clause specifies the structure of a menu after selection:

$offeritems(commandmenu) :-$
$part(commandmenu, monitoritem), part(commandmenu, restartitem),$
$part(commandmenu, manualitem), part(commandmenu, offitem),$
$not(part(commandmenu, label))).$

This states that the items of *commandmenu* are available for selection if *monitoritem*, *restartitem*, *manualitem* and *offitem* are parts of *commandmenu* and *label* is not part of *commandmenu*. Intuitively, this requirement could not be satisfied without an inconsistency because the initial specification of the menu stated that *label* was a part of *commandmenu*. Such problems can be avoided by maintaining a database which records the current structure and image of an interface. For instance, if a user pulled down *commandmenu* then the database must be updated to show that *label* is no longer part of the structure of the menu. Schappo and Edmonds [20] support the maintenance of graphical records by providing meta-predicates, *assertg* and *retractg*. There are a number of technical problems which surround the use of such predicates [15].

3. FIRST ORDER LOGIC AND DIALOGUE SPECIFICATION

First order logic provides a means of structuring descriptions of complex graphical images. Events provide a means of incorporating these structures into specifications of interactive dialogues. Input from a range of physical devices, such as mice or tracker balls, can be represented by generic events, such as *select* or *move*. For example, a designer might construct a prototype in which a command to turn a system off was successful if it had an appropriate effect and this was displayed:

$resolveerror1 :-$
$display(errordisp), subpart(errordisp, offitem), select(offitem),$
$effect(select(offitem), error, off), display(offdisp).$

An error is resolved if *errordisp* is displayed and *offitem* is selected and *offitem* is part of *errordisp* and the effect of the selection is to turn a system from the *error* to the *off* states and this effect is displayed. Such specifications leave the designer free to explore the "design space of input devices" by using a variety of devices to generate the *select* event [3].

The lack of sequencing in first order logic poses particular problems for event based specifications of interactive dialogues. The previous clause would be satisfied if *select* were received after it had taken effect. The specification does not determine the order in which *display(errordisp)* or *display(offdisp)* are to be evaluated. In order to implement such clauses, logic programming languages must enforce an order of evaluation. For example, the left-right strategy of PROLOG would display *errordisp* before the selection event was received, the input event would take *effect* before *offdisp* was presented. This introduces a distinction between the logic of a specification and the control which implementations require in order to determine an order of evaluation [13]. An implementation of the previous clause would only exhibit the 'desired' behaviour because control considerations have influenced the ordering of the logical specification. Changing the order of *select(offitem)* and *display(offdisp)* would have no effect on the specification of a dialogue but would radically affect the behaviour of a prototype. This introduces considerable complexity into the refinement necessary to implement high level specifications. The

ordering of clauses within a logic specification may have to be substantially revised to support the control strategy of an implementation. Alternatives must be sought if logic programming is to provide a convenient bridge between prototyping and specification.

4. TEMPORAL GRAPHICS

Many of the limitations described in previous sections may be avoided by the introduction of temporal information into logic specifications of interactive systems. Dialogue sequences and changes in the structure of a graphical interface may be described with respect to an interval of time. Such information can be introduced in a number of ways. For example, Sundgren describes 'elementary' records which associate time stamps with every clause in a specification [21]. Dialogue sequencing can be made explicit within the logic of the specification. This removes any reliance upon the control strategies implicit within logic programming languages:

$resolveerror2 :-$
$display(errordisp, 120005), select(offitem, 120010),$
$effect(select(offitem), error, off, 120015), display(offdisp, 120020).$

This states that an error is resolved if *errordisp* is displayed at five seconds past midday and *offitem* is selected five seconds later and this has the effect of turning an erroneous system *off* a further five seconds later and this effect is displayed at exactly twenty seconds past midday. Unfortunately, this approach can be unwieldy for non-trivial specifications. The expression of complex timing properties would require the introduction of a large number of temporal parameters. Specifications which describe persistent properties of interaction are especially cumbersome to construct using explicit time stamps. For example, the following clause requires that *errordisp* is displayed every second until the user responds with an appropriate input:

$resolveerror3 :-$
$display(errordisp, 120005), display(errordisp, 120006),$
$display(errordisp, 120007), display(errordisp, 120008),$
$display(errordisp, 120009), select(offitem, 120010),$
$effect(select(offitem), error, off, 120015), display(offdisp, 120020).$

Specifications constructed using fixed time stamps impose unrealistic demands upon system operators. In order to fulfill the previous requirement a user must provide input at exactly ten seconds after midday! This is likely to be an extremely optimistic assumption given finite cognitive resources and divided attention in noisy environments. Such determinism can be avoided by substituting temporal variables for explicit parameters:

$resolveerror4 :-$
$display(errordisp, T), select(offitem, T1), effect(select(offitem), error, off, T2),$

$display(offdisp, T3), after(T, T1), after(T1, T2), after(T2, T3).$

This states that an error is resolved if *errordisp* is displayed at moment T and *offitem* is selected at $T1$ and this has the effect of turning an erroneous system *off* at $T2$ and this effect is displayed at $T3$ and $T1$ occurs after T and $T2$ after $T1$ and $T3$ after $T2$. The temporal ordering within the clause is made explicit by the predicate *after*. Unfortunately, the use of temporal variables still involves the designer in considerable complexity [6]. In particular, the designer is responsible for maintaining the semantics of predicates, such as $after$, which define an orderings over variables. These semantics can radically effect the properties of any specification [19].

Temporal logic provides the interface designer with a means of avoiding the complexity associated with time stamps. This formalism extends first order logic to include the following operators: $\Diamond$ (read as 'eventually'); $\bigcirc$ (read as 'next'); $\Box$ (read as 'always') and $\mathcal{U}$ (read as 'until'). It provides the designer with well developed proof techniques through Kripke semantics [14]. The use of temporal logic relieves the designer from the burdens of maintaining an explicit ordering in terms of predicates such as $after$. The ordering is captured within the definition of temporal operators. For example, $\Diamond$ may be defined using a set of time stamps T, $|w|_t$ denotes the truth value of the formula w at time t:

$|\Diamond(w)|_t \equiv \exists t1 \in T[after(t, t1) \wedge |w|_{t1}]$

Informally this states that w is eventually true at time t if it is true at a time after t. The other temporal operators can be defined in a similar fashion. A more complete introduction is omitted for the sake of brevity and the interested reader is directed to Prior [18]. The encapsulation of sequencing within definitions of temporal operators provides a tractable means of including dynamic information within declarative specifications of graphical interfaces.

5. TEMPORAL LOGIC AND DECLARATIVE GRAPHICS

Higher order, temporal logic, provides a precise and concise means of describing structural changes in part-whole hierarchies. It can be used to avoid the axiomatic approach which led to a contradiction in first order logic. For example, a designer can specify that eventually a pull down menu is represented by a label or command options:

$\Diamond(part(commandmenu, label))$.
$\Diamond(part(commandmenu, offitem), part(commandmenu, onitem))$.

This clause does not require that the structure of *commandmenu* must always include *label*. It is, therefore, possible that at some time the structure of *commandmenu* may not include *label*:

$changestructure(commandmenu) : -part(commandmenu, label),$
$select(label), \bigcirc(part(commandmenu, offitem),$
$part(commandmenu, monitoritem), part(commandmenu, restartitem),$
$part(commandmenu, manualitem), not(part(commandmenu, label))).$

This states that there is a change in the structure of *commandmenu* if in the present interval *label* is part of *commandmenu* and there is a selection event for *label* and in the next interval *offitem*, *monitoritem*, *restartitem* and *manualitem* are parts of *commandmenu* but *label* is not.

Temporal operators provide a means of describing structural changes which are often used to indicate command completion in graphical interfaces [7]. For example, many systems confirm the successful completion of a command by hiding the options of a pull down menu. In order for the user to predict the success of a command in such an interface it should always be the case that the initial structure of the menu should eventually return once it has been pulled down. Such requirements are not easily expressed using static, declarative, definitions of part-whole hierarchies. The $\Box$, $\bigcirc$ and $\Diamond$ operators provide a tractable means of expressing such dynamic requirements:

$\Box(dialoguecycle : -changestructure(commandmenu),$
$\bigcirc(select(offitem), part(commandmenu, offitem),$
$\Diamond(effect(select(offitem), error, off), part(commandmenu, label)))).$

Expressing this requirement in terms of fixed time stamps or temporal variables is a non-trivial task.

6. TEMPORAL LOGIC AND DIALOGUE SPECIFICATION

Sequencing information can be incorporated into a logical specification using the $\bigcirc$ operator. For example, a designer might specify that the presentation of *errordisp* should immediately be followed by input to turn the system off, that the effect of this input should be followed by the presentation of $offdisp$:

$resolveerror5 : -display(errordisp),$
$\bigcirc(select(offitem), effect(select(offitem), error, off)), \bigcirc(display(offdisp))).$

This states that an error is resolved if in the present interval *errordisp* is displayed and in the next interval *offitem* is selected and the effect of this is to turn the system off and this is displayed in the next again interval. The inclusion of temporal operators makes the sequence of a dialogue explicit within the logic of a specification. This provides a means of expressing the dialogue requirements which a user must be able to satisfy in order to guarantee the success and safety of an application. They must be able to recognise the *errordisp* and then respond appropriately by turning the system off. The use of temporal operators also avoids any reliance upon the evaluation strategies

implicit within the control of an implementation. The behaviour of an interface would not be affected if the ordering of a clause were changed. Unfortunately, this technique suffers from some of the problems associated with time stamps. Both approaches are strongly deterministic, each predicate is evaluated during a specific interval in time. The $\Diamond$ operator provides a means of avoiding such determinism:

$resolveerror6 : -display(errordisp),$
$\Diamond(select(offitem), effect(select(offitem), error, off), \bigcirc(display(offdisp))).$

This states that an error is resolved if in the present interval *errordisp* is displayed and eventually *offitem* is selected and the effect of this is to turn the system *off* and in the next interval this is displayed. The designer may not be able to predicting the exact moment at which an operator will react to the presentation of *errordisp*. The introduction of the $\Diamond$ operator captures this uncertainty by specifying that a user should eventually respond. The designer can use such operators to describe the influence of finite cognitive resources and noisy working environments which prevent users from immediately fulfilling dialogue requirements.

The $\bigcirc$ and $\Diamond$ operators provide a means of integrating dynamic requirements into a specification that supports proof. They do not, however, address the issue of persistence which was a significant drawback to the use of fixed time stamps, illustrated by $resolveerror3$. The previous section has described how the $\Box$ operator provides a solution to this problem. The designer is not required to explicitly restate axioms of a specification for each interval. Temporal logic also provides a means of describing bounded persistence. Transient displays are easily forgotten or overlooked. Previous clauses might be refined so that *errordisp* is presented until input is received. The $\mathcal{U}$ operator (read as 'until') provides a means of specifying such requirements:

$resolveerror7 : -display(errordisp)\mathcal{U}$
$(select(offitem), effect(select(offitem), error, off), \bigcirc(display(offdisp))).$

This states that an error is resolved if *errordisp* is displayed until *offitem* is selected and the effect of this is to turn the system *off* and in the next interval this is displayed. Temporal logic provides an expressive means of specifying changes in graphical structures and dialogue sequences for dynamic systems. In order to benefit from the integrated approach advocated in the introduction there must be some means of deriving prototype implementations from such specifications.

7. IMPLEMENTATION

A central tenet of our previous work has been that specifications and prototyping must be integrated to support interface design. PRELOG, a tool for Presenting and REndering LOGic specifications of interactive systems, has been implemented. This links the tractability and executability of PROLOG with a screen presentation system,

Presenter [23]. PROLOG provides a means of prototyping high level specifications using

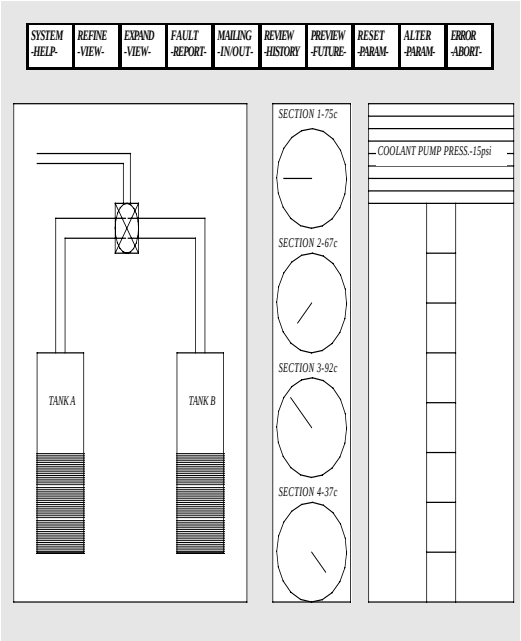

Figure 3: Part of an early PRELOG prototype.

an executable subset of first order logic. Presenter handles low level device primitives and supports the implementation of device abstractions in the form of input *events*. PRELOG has been implemented on a network of Sun 3/50s and 3/60s. The development of this tool has revealed many of the limitations of using first order logic to design dynamic graphical interfaces for complex systems. A reliance upon extra-logical control facilities in dialogue specifications and the use of internal databases to maintain records of graphical structures has increased the difficulty of the transformation between specification and prototype. It is possible to overcome many of these limitations by extending PRELOG to provide access to a temporal logic interpreter.

Several research groups have attempted to provide an executable semantics for temporal logic specifications. Hale and Moszkowski have produced interpreters for the Tempura temporal logic programming language [16]. Current implementations are written in C and Lisp and could not easily be incorporated into the declarative graphics system supported by PRELOG. The meta-interpreter Tokio [1] did not suffer from this disadvantage. Written in PROLOG it proved a trivial task to extend PRELOG to include the temporal operators provided by Tokio. Figure 4 illustrates the resulting system architecture. If there are no temporal operators in a clause then Tokio will pass it directly on to PROLOG for evaluation. Clauses which contain temporal operators are rewritten in first order logic and asserted over an appropriate interval. The interpreter maintains the fixed time stamps which otherwise might impose heavy burdens upon the designer. Execution progresses when all the clauses for a particular interval are satisfied.

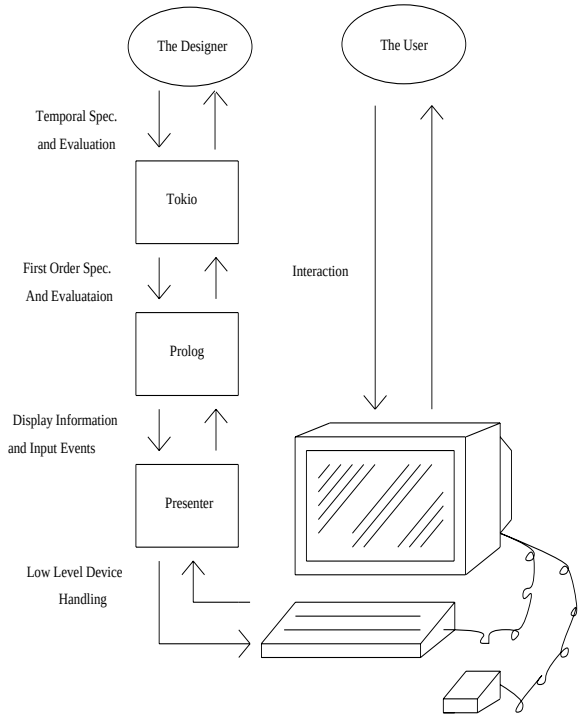

Figure 4: The extended PRELOG architecture.

8. CONCLUSION AND FURTHER WORK

Brevity has forced the omission of an extended example of the application of temporal logic to support interface design. Similarly, it has not been possible to include a detailed exposition of the proof system which supports the temporal extensions to PRELOG. These have been addressed elsewhere and the interested reader is directed to [9, 11]. Instead, this paper has focussed upon the problems of temporal reasoning using first order logic. It has been argued that these must be resolved if this formalism is to support the design of dynamic graphical interfaces. Previous systems have exploited control strategies and extra-logical facilities available within logic programming languages, such as PROLOG. This forces the introduction of implementation issues into the logic of a specification and incurs an additional complexity for the refinement necessary in order to render a specification executable. Higher order, temporal, logic has been proposed as an elegant alternative to these ad hoc solutions. The sequence of a dialogue can be made explicit within a specification. The structure and appearance of an image can be quantified with respect to time; static, declarative, specifications can be animated.

Future work will continue to explore the use of temporal logic and structural de-

composition to support the design of complex graphical interfaces. In particular, it is hoped that these techniques will provide valuable tools for the design of interfaces to process control systems. Temporal logic is appropriate for this investigation because timing properties are often critical to the safe operation of such dynamic applications. It is also hoped to extend PRELOG to include some of the facilities offered by constraint based graphics languages [22]. Future work will explore the relationship between the specification of weak constraints, involving the $\Diamond$ operator, and the hard commitments necessary for execution, described using the $\bigcirc$ or $\Box$ operators. Initial investigations have also revealed that this approach has some potential as a means of supporting model-based image recognition [4]. It is hypothesised that the introduction of temporal dependencies will significantly increase the power of current modelling techniques for image analysis.

ACKNOWLEDGEMENTS

Thanks are due to the members of the Human Computer Interaction Group at York who provided valuable encouragement in this research. This work has been supported by a CASE award funded by British Telecom, by SERC grant 88503497 and by the 1990 Gibbs-Plessey travel award.

REFERENCES

1 T. Aoyagi, M. Fujita, and T. Moto-Oka. Temporal logic programming language-TOKIO-programming in TOKIO. In E. Wada, editor, *Logic Programming '85*. Springer Verlag, Berlin, FDR, 1986.

2 E.H. Blake and S. Cook. On including part-hierarchies in object-oriented languages. In *ECOOP '87 European Conference On Object Oriented Programming*, volume LNCS 276, pages 41–50, Berlin, F.D.R., 1987. Springer Verlag.

3 S.K. Card, J.D. Mackinlay, and G.G. Robertson. The design space of input devices. In *CHI '90*, pages 117–124. Addison Wesley, Reading, MA, United States Of America, 1990.

4 R.T. Chin and C.R. Dyer. Model-based recognition in robot vision. *Computing Surveys*, 18(1):67–108, March 1986.

5 K. L. Clark and J. Darlington. Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65, February 1980.

6 D.M. Gabbay. Modal and temporal logic programming. In A. Galton, editor, *Temporal Logics And Their Applications*, pages 197–223. Academic Press, London, United Kingdom, 1987.

7 M. D. Harrison, C. R. Roast, and P. C. Wright. Complimentary methods for the iterative design of interactive systems. In G. Salvendy and M.J. Smith, editors, *Designing And Using Human-Computer Interfaces And Knowledge Based Systems*, pages 651 – 658. Elsevier Scientific, North Holland, 1989.

8 C.W. Johnson. Using temporal logic to prototype interactive systems. In G. Cockton D. Diaper, D. Gilmore and B. Shackel, editors, *Interact '90*, pages 1019–1021. Elsevier Science, Amsterdam, The Netherlands, 1990.

9 C.W. Johnson. *Predictability And The Temporal Logic Of Interactive Control Systems*. PhD thesis, Department Of Computer Science, University of York, York, United Kingdom, 1991 (in preparation).

10 C.W. Johnson and M.D. Harrison. PRELOG-a system for presenting and rendering logic specifications of interactive systems. In C.E. Vandoni and D.A. Duce, editors, *Eurographics '90*, pages 469–480. Elsevier Science, Amsterdam, The Netherlands, 1990.

11 C.W. Johnson and M.D. Harrison. Using temporal logic to support the design and implementation of interactive control systems. Technical report, (Accepted for the International Journal Of Man-Machine Studies) Department Of Computer Science, University of York, York, United Kingdom, 1990.

12 H.J. Komorowski and J. Maluszynski. Logic programming and rapid prototyping. *Science of Computer Programming*, 9:179–205, 1987.

13 R. Kowalski. Algorithm = logic + control. *Communications Of The A.C.M.*, 22(7):424 – 436, July 1979.

14 Z. Manna and A. Pnueli. Verification of concurrent programs: Temporal proof principles. In D. Kozen, editor, *Logic Of Programs 1981*, pages 200–252. Springer Verlag, Berlin, F.D.R., 1982.

15 C. Moss. Cut and paste : Defining the impure primitives of PROLOG. In E. Shapiro, editor, *Third International Conference On Logic Programming*, pages 686 – 694. Springer-Verlag, Berlin, F.D.R., 1986.

16 B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, United Kingdom, 1986.

17 F. C. N. Pereira. Can drawing be liberated from the Von Neumann style? In M. Van Caneghan and D. H. D. Warren, editors, *Logic Programming And Its Application*, pages 175 – 187. Ablex Publishing, Norwood, New Jersey, United States of America, 1986.

18 A. Prior. *Past, Present, Future*. Oxford University Press, Oxford, United Kingdom, 1967.

19 N. Rescher and A. Urquhart. *Temporal Logic*. Springer Verlag, Vienna, Austria, 1971.

20 A. Schappo and E. A. Edmonds. Support for tentative design : Incorporating the screen, as a graphical object, into PROLOG. *Int. Journal Of Man Machine Studies*, 24:601 – 609, 1986.

21 B. Sundgren. Conceptual foundations of the infological approach to data base management. In *Proceedings IFIP Conference On Data Base Management*, pages 61–94, 1974.

22 P. Szekely and B. Myers. A user interface toolkit based on graphical objects and constraints. *Journal of A.C.M. SIGPLAN Notices*, 23(11):36–45, November 1988.

23 R. Took. Surface interaction a paradigm and model for the presentation level of applications and documents. In J.C. Chew and J. Whiteside, editors, *CHI '90*, pages 35–42. ACM Press, New York, The United States of America, 1990.