Programming Research Group

USER INTERFACE LANGUAGES: A SURVEY OF EXISTING METHODS

Gregory Abowd
Jonathan Bowen
Programming Research Group
Oxford University

Alan Dix
Michael Harrison
Roger Took
Department of Computer Science
University of York

October, 1989

PRG-TR-5-89

Oxford University Computing Laboratory
11 Keble Road, Oxford OX1 3QD

User Interface Languages: a survey of existing methods*

Gregory Abowd
Jonathan Bowen†
Programming Research Group
Oxford University

Alan Dix‡
Michael Harrison
Roger Took
Department of Computer Science
University of York

October, 1989

Abstract

This report gives a survey of user interface languages and formal representations of user interfaces. The following aspects of User Interface Languages are considered:

- expressiveness
- readability
- evaluation (is it possible to evaluate the ergonomic and functional quality of the user interface from the representation)
- manipulation
- compilation/interpretation
- possibility to include knowledge representation.

---

*Further copies of this Technical Report may be obtained from the Librarian, Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, England (Telephone: +44-865-273837, Email: `library@comlab.ox.ac.uk`).

†Research Officer, SERC Software Engineering Project, Programming Research Group, Oxford University

‡SERC Post-doctoral Fellow, Human Computer Interaction Group, Department of Computer Science, University of York

Contents

Introduction

This document is a survey of formal notations, models and techniques for specifying the end user interface of computer systems. These vary from highly user-oriented psychological models to highly computer-oriented implementation architectures, from very abstract models of systems to concrete key-by-key descriptions.

Chapter 1 briefly looks at a wide range of such notations. Chapters 2 and 3 reflect backgrounds of the authors, the first describing the range of abstract mathematical models of interactive systems studied at York University and the second rendering one of these models in the *Z* specification language and then discussing a related model developed at the Programming Research Group (PRG) in Oxford.

Chapter 4 discusses some of the factors that should be considered when comparing and evaluating the various notations. This includes an examination of the niches within which the different notations fit and the trade-offs between different desirable features. The following chapter 5 gives a summary evaluation of the various classes of notation introduced in Chapter 1 against the criteria established for this report:

- expressiveness
- readability
- evaluation
- manipulation
- execution
- knowledge

Finally, Chapter 6 makes broad recommendations in the light of the desire for reverse engineering and the specific areas of

- task representation
- dialogue specification
- statement of high level requirements and standards

In an appendix is a section from Roger Took's thesis [133] which discusses some of the vast range of techniques for input parsing, used primarily in UIMS and window managers which seemed too detailed to include in the main body of the report.

Chapter 1

Description of notations

This Chapter describes briefly a range of notations arranged loosely into the types:

- Psychological and soft computer science notations
- User models
- Graphical or diagrammatic approaches
- Abstract mathematical models
- General purpose formal notations
- Formal dialogue specifications
- Non-committed prototyping notations
- Dedicated interface prototyping and development tools
- User Interface Management Systems (UIMS) and Window managers

Of course, no such classification scheme is quite right and some notations fit into several categories. Chapter 4 gives alternative ways of classifying the notations. On the whole, each of the classes given can be placed into a single one of the dimensions described there, and have roughly similar summary evaluations.

1.1 Psychological and soft computer science notations

These are formalisms which have been developed by psychologists, or computer scientists whose interest is in understanding user behaviour. A useful summary of these formalisms may be found in the paper by Green, Schiele and Payne [41] who classify them in respect to how well they describe features of the *competence* and *performance* of the user. A *task* or *goal* is basic to both approaches. In practice all the notations that deal with competence and performance incorporate aspects of both to a greater or lesser degree. Quoting from Simon [118]:

> Competence models tend to be ones that can predict legal behaviour sequences but generally do this without reference to whether they could actually be executed by users. In contrast, performance models not only describe what the necessary behaviour sequences are but usually describe both what the user needs to know and how this is employed in actual task execution.

Simon goes on to classify these notations (and cognitive models in general) in a 3-dimensional space, representing various trade-offs made by their designers.

A more computational classification of these notations would be as:

- Hierarchical representation of the user's task and goal structure
- Describing the dialogue as a language (formal grammar)

Representative of the former school would be *GOMS*[1] [12]. This assumes the user has a hierarchical structure of goals, and subgoals. The sub-goal decomposition may be deterministic or may involve choice among different strategies for achieving the goal. At the leaves of the resulting goal tree are the operations that the user carries out to achieve the lowest level of goal. The analysis can be carried out at various levels of granularity, depending on what operations are regarded as basic. So for example at a course level "edit document" may be regarded as basic, whereas for finer grained analysis "press the cursor up key" may be terminal. By analysing the goal structure, putative measures of performance can be given. These use things like stacking depth of goals in order to estimate for instance short term memory requirements. The models of the users' mental processes implied by this are very idealised.

Representative of the *linguistic approach* would be Reisner's use of *BNF* type rules to describe the dialogue grammar [102]. This views the dialogue at a purely syntactic level, ignoring the semantics of the language. Typically grammar rules ignore computer output and this emphasis on the complexity of input is widespread. (It is easy to print, but possibly difficult to parse.) Others have added actions to grammar rules, which include output or (more uniformly) have included output as well as input among the grammar's terminals. There are well known techniques for analysing the complexity of grammars, and these can be used to give a crude measure of the difficulty of a dialogue, however the interpretation of such measures is severely complicated by such things as familiarity with (perhaps complex) grammatical forms, clear mode changes etc.

TAG (task action grammar [98]) tries to deal with some of these points by including elements such as parameterised grammar rules to emphasise consistency and world knowledge (e.g., up is the opposite of down). For example TAG could be used to represent the user's knowledge of how to draw a graphic object in Apple MacDraw:

How to draw a rectangle

> select rectangle tool, place mouse at one corner of the desired rectangle, depress button, drag to opposite corner, release button.

This notion may be generalised in TAG notation and hence give some understanding of the consistency within the specification.

Draw a rectangle or square

$$
\begin{aligned}
&task[effect = add, type\ of\ object = rectangle, constraint = any, selecttool = any] :=\\
&\quad selecttool[type\ of\ object = rectangle] + draw[Constraint]\\
&draw[Constraint = yes] := press\ SHIFT + place\ mouse \ldots\\
&draw[Constraint = no] := place\ mouse \ldots
\end{aligned}
$$

Within this claimed mental representation of the system it then becomes possible to analyse notions of consistency. Here consistency is related to the user's understanding. Consequently

[1] Goals Operation Methods Selection.

there are clear design implications. What TAG does not attempt to do is to provide clear linkage between appropriate Task Action Grammars and design.

Recent developments of the notation (by Hayes and Payne at Lancaster) include attempts to make good some of the limitations of TAG, in particular to include display information and flow information (no state is implied by TAG). A pessimistic view of these developments is that they make an already cumbersome notation worse. The possibility of developing informative notions of consistency becomes even more remote. Although this notion is purported to be a competence formalism [41], it is clear that the breakdown of the task into action also has performance implications. Although there have been some attempts to scale up TAG to quite large applications these activities have not been satisfactory.

Grammars are of course easy to execute, there being many standard parser generators. *Yacc* [64] (the standard UNIX parser generator) handles interactive dialogue tolerably well, and the field is so well understood that writing specialist tools is no great problem, the only possible worry being that good grammars for parsing (*LL/R(1)*) may not necessarily be best for usability. A second problem is error recovery, which is difficult enough in batch systems, but is clearly more important in interactive systems. Insufficient emphasis on imperfect dialogue is common among all notations.

An execution of a grammar may be used as a simple prototype of the system. It will enable the designer to experiment with different input dialogues. If the grammar includes actions or system responses then these will be part of the prototype. Clearly some prototype of the application will be required if more sophisticated feedback is required.

If the grammar is intended to model the user's goal structure, then executing the grammar to obtain parse trees of various input dialogues would enable certain crude cognitive measures to be made. Each non-terminal would correspond to a sub-goal, and hence the depth of the parse tree would correspond to the depth of the user's goal stack.

Taking their lead from linguistic theory and parsing, it is often suggested that several levels of grammars ought to be used. Moran's *CLG* [85] is probably the most well known concrete example of this. It uses four levels: lexical, syntactic, semantic and finally task level. It is more design oriented than most other similar approaches, the task level being described first, obtained presumably from a task analysis, then the semantic level, formalising the entities, before moving on to the more concrete levels. Various rules are given for checking consistency within and between the levels, although these are rather loose and incomplete. This approach comes closer to viewing the entire system as involved in the interaction rather than just the surface dialogue. Unfortunately, it has been found unwieldy to use in practice [110, 111], and the notation used is (among arcane notations) particularly arcane having a very LISP-like flavour.

CCT (cognitive complexity theory [69]) combines the goal hierarchy and dialogue grammar approaches. It has two parallel descriptions, a GOMs like one, using *production rules*, for the user goals and *GTNs* (generalised transition networks) to describe the system grammar. The production rules are a sequence of rules:

if condition *then* action

where *condition* is a statement about the contents of working memory. If the condition is true then the production is said to have fired. An *action* may consist of one or more elementary actions. The "program" is written in a LISP-like language and generates actions at the keystroke level that have associated performance characteristics. A typical program fragment [69] would be:

```
IF (AND (TEST-GOAL delete word)
```

```
         (TEST-CURSOR %UT-HP %UT-VP))
THEN (   (DO-KEYSTROKE DEL)
         (DO-KEYSTROKE SPACE)
         (DO-KEYSTROKE ENTER)
         (WAIT)
         (DELETE-GOAL delete word)
         (UNBIND %UT-HP %UT-VP )))
```

This "user program" can be executed and assessed empirically and analytically. In addition, mismatches between it and the system grammar can be found and a dissonance measure produced. The GTNs which describe this system grammar are in the form of diagrams representing the dialogue states with arcs representing the possible transitions on user actions. The difference from simple state transition diagrams is that the nodes may be hierarchically decomposed. This system part of CCT could be executed in the same way as a grammar to give a crude dialogue prototype.

The formation of the goal hierarchy is largely a post-hoc technique and runs a very real risk of being defined by the dialogue rather than the user. Knowles [71] attempts to rectify this by producing a goal structure based on a pre-existing manual procedure, she thus hopes to obtain a natural hierarchy. In addition, she criticises the mechanical measures of complexity because they do not take into account issues such as user knowledge. She goes on to produce a more subjective analysis, using the mismatches highlighted, but incorporating more expertise. The production rules, of course, form an executable dialogue description, but their intention is descriptive.

There is a worry that grammar based techniques are not good at describing more modern windowed and mouse driven interfaces, being better suited to command based or at least keystroke based dialogues. One problem here is the lowest level lexical structure. Pressing a cursor key is a reasonable *lexeme*, but moving a mouse one pixel is less sensible. In addition, pointer based dialogues are more display based. Clicking a cursor at a particular point on the screen has a meaning dependent on the current screen contents. This problem can be partially resolved by regarding operations such as "select region of text" or "click on quit button" as the terminals of the grammar. If this approach is taken, the detailed mouse movements and parsing of mouse events in the context of display information (menus etc.) are abstracted away. This of course means that any prototyping of the dialogue will be at a similarly abstract level or require "Wizard of Oz" techniques to mock up the full interface.

Goal hierarchy methods have different problems, as more display oriented systems encourage less structured methods for goal achievement. Instead of well defined plans, the user is seen as performing a more exploratory task, recognising fruitful directions and backing out of others. Typically even when this exploratory style is used at one level we can see within it and around it more goal oriented methods. So for example, we might consider the high level goal structure:

```
WRITE_LETTER   ==>   FIND_SIMILAR_LETTER + COPY_IT + EDIT_COPY
```

However, the task of finding a similar letter would be exploratory, searching through folders recognising possible places would not be well represented as a goal structure at all. Similarly the actual editing would depend very much on non-planned activities: "ah yes, I want to re-use that bit, but I'll have to change that". If then, we drop to a lower level again, goal hierarchies become more applicable, for instance, during the editing stage we might have the (classic) delete a word sub-dialogue:

```
DELETE_WORD   ==>   SELECT_WORD + CLICK_ON_DELETE
```

```
SELECT_WORD    ==>    MOVE_MOUSE_TO_WORD_START + DEPRESS_MOUSE_BUTTON
                      + MOVE_MOUSE_TO_WORD_END + RELEASE_MOUSE_BUTTON
CLICK_ON_DELETE  ==>  MOVE_MOUSE_TO_DELETE_ICON + CLICK_MOUSE_BUTTON
```

Thus goal hierarchies can partially cope with display oriented systems by appropriate choice of level, but the problems do emphasise the rather prescriptive nature of the cognitive models underlying them.

1.2 User models

CCT is interesting in that it encompasses two alternative views of the dialogue, one a user based goal structure, the other the system based grammar. The GOMs part can be seen as a rather rudimentary *user model*.[2] This process of modelling the user can be carried out at a more sophisticated level. Some approaches, such as *SOAR* [72], are at the moment primarily interested in understanding human cognition and are not aimed at design. Often many of the grammar and goals oriented techniques are included within this general umbrella, including GOMs, *MHP* (model human processor [12]), *TAG*, and *KLM* (keystroke level model [11]).

A more recent strand has been the investigation of *programmable user models*. Work at APU Cambridge [144, 143], for instance, includes executing programs in the SOAR cognitive architecture to perform *scenarios* (typical examples of user interaction with the machine).

All these approaches are still at the research stage. However the general idea that producing a description of how the user is to accomplish expected tasks in parallel to the actual system development seems useful. It is generally agreed that the form of the modelling is not nearly as important as the discipline it enforces on the designer.

1.3 Graphical or diagrammatic approaches

On the principle that many people (especially the less formally minded) find graphical notations easier to use, there have been many different notations proposed. Obviously most of the hierarchical and grammar notations can be given a graphical form, and in addition there are data-flow diagrams, state transition diagrams (of many flavours), *JSD*[3] diagrams and simple flow diagrams.

Diagrammatic notations are often used in conjunction with other notations and may have automatic support. For instance, Marshall's diagrammatic notation [77] (see below) links to VDM[4]. England has built graphical design tools [31] for state transition based dialogues as part of the Alvey *Eclipse project*, which links in to the Eclipse screen format definition language (*FDL* [30]) described below in Section 1.8. He shows that the notation helps highlight inconsistencies within the dialogue.

Sutcliffe [126] has used JSD process structure diagrams to describe tasks. He then analyses these in order to highlight possible problems such as memory limitations (rather like GOMs). Similarly Walsh *et al.*[139] have integrated task analysis techniques with JSD, they point out that these notations are already heavily used for the software development side, and therefore they form a common language. JSD diagrams can be used quite simply as a model of the dialogue, being a particular form of grammar.

By way of illustration, (but not recommendation!), I have myself used simple flow diagrams in order to specify dialogues for transaction processing systems. These were systematically (but

[2] User model is used in several contexts, but here it has the meaning of "designer's model of the user".

[3] Jackson System Development.

[4] Vienna Development Method.

manually) translated into COBOL programs (essentially a program inversion exercise). Finally, screens painted by the intended users were linked in. These prototypes (which were in fact fully running systems) could be produced within hours or at worst days. Thus within a restricted domain, even quite simple specification tools can be highly productive.

1.4 Interaction models

These were developed at York for the expression of user interface properties. They are not a notation in themselves, being expressed over standard mathematical set theory. They have been rendered into specific formal notations both at York and elsewhere. These can be used early in design to decode the general shape and character of the interface and to determine which qualities are required.

Typical properties that are described using interaction models are the ability to reach any point in the dialogue, or the reversibility of dialogue steps. They also stress the relation between the system's display and its internal state, expressing the predictability of the system behaviour given the user's view.

The earliest, and most general of these models (the *PIE model* [28]) has been used in the design of prototype systems using algebraic specification at York, incorporated into a design technique using formal grammars and denotational semantics by Anderson [6], and rendered by Sufrin [125] into the *Z* notation. In addition, Monk [83] produced an analysis and design notation *Action Effect* rules, which is less formal and applies to a more restricted class of systems, but which is more amenable to use by human factors practitioners. Runciman [107] has developed a prototyping approach for purely functional programs based around the PIE model.

As this is a York speciality, it will be described in greater detail in Chapter 2.

1.5 Using general purpose formal notations

Notations intended for general software design have been used to specify interactive systems. Examples include Sufrin's elegant specification of a text editor using Z [123], and a similar specification by Ehrig [29] in the *algebraic specification* notation *ACT-ONE*. Chi produced no less than four specifications of an editor in different notations in order to compare their utility, although the varying skill with which he used the different styles somewhat invalidates his results. At York University an experimental multi-window editing environment was specified and implemented, which allowed different formatting styles and provided some crude hyper-text facilities [23]. This was done in an algebraic framework, but with a much sugared notation compared with (for example) ACT-ONE.

Various interface components have also been described. Took [132] fully specified his *Presenter* display manager in Z. Window based screens have been defined both in *ADT* (an algebraic language) and Z [10]. Cook [18] describes how generic interface components can be specified in a purely functional language.

1.6 Formal dialogue specifications

To some extent, the problem with the use of general specification techniques, is that they are too general. The dialogue component as such will not be clear, and has to be "modelled" in the notation. Sometimes the required special "interaction" forms of use can be packaged up, for instance by using higher level functions in a pure functional setting. These may then be used within the standard notation, allowing the free add mixture of interface and standard forms. This gives maximum expressivity, but at the cost of losing the dialogue/application separation which is frequently seen as desirable. In particular it makes it hard to analyse the dialogue structure as a separate part of the system.

There have been various attempts to add dialogue specification components to standard notations. These may be simply sugarings that are then translated into the underlying notation to give them semantics, or have a separate level of semantics given them. In either case, the actual concrete notation makes a clear separation between the two styles of specification.

Hekmatpour and Ince [51], for instance, have a separate user interface design component in their specification language *EPROL* (or strictly *wide-spectrum language*). This interface component seems rather disappointing however, being simply a 'teletype' forms and menu description such as may be included with many data-base languages or *4GLs*[5]. The dialogue is apparently described entirely within the main specification language and may thus be easily obscured.

Marshall [77] has merged a graphical dialogue specification technique with *VDM*. This includes standard constructs such as sequence, choice and iteration in the dialogue, each terminal dialogue "box" is related to a piece of VDM specification. She also suggests that the user's actions in this can be represented by a parallel diagrammatic/VDM description, but in her examples this diagram consists of a single box, so the claim is a trifle premature. This exposes the fact that the diagrammatic notation does not support parallel activities (such as multi-window dialogues). It would be quite easy to add such a construct at the diagrammatic level, but the meaning when translated into VDM semantics would not be clear. The actual acceptance of input is handled by "shared" global variables with the user "process" and is hardly clean. Another problem, is that each piece of VDM works on global variables, making it difficult to trace the semantic impact of particular user actions without analysing the pieces of VDM in detail (hardly dialogue separation).

Alexander [4] has designed an executable specification/prototyping language around *CSP* and *me-too* called *SPI*, (specifying and prototyping interaction). This has several similarities to the above. Me-too is an executable specification language based on VDM and implemented under several dialects of LISP. The CSP forms the dialogue specification part, whereas the me-too supplies the semantics. This is rather similar to the way *CCS* (a CSP-like notation) and ACT-ONE are combined in *LOTOS*.

SPI's dialogue component is called *eventCSP*, it includes most CSP constructs, sequential action, choice, iteration, and most importantly parallel composition. The parallel composition makes it possible to express concepts such as the choice between mouse and keyboard input. The expression of choice is based on the occurrence of events and is thus more clear. It inherits drawbacks from CSP however, such as the lack of direction in events, it is not evident in the syntax whether an event is due to external input, produces external output or is an internal synchronisation between parallel processes. This can be confusing in dialogue design when there is an obvious direction of control flow. however, the problem is largely mitigated for user I/O by the judicious choice of event names. It is thus only internal events that remain confusing. The structure of possible events is static too. This would make it hard to deal with the dynamic creation of windows for instance. This lack of dynamic configuration (and related lack of parameterisation) is common to many dialogue languages, it would be easy to add to most, but would typically reduce the possibility of analysis of the dialogue structure.

The semantic part of SPI is called *eventISL*. Although I have said it is based upon me-too, this is in fact its first "host" language and it is intended to operate with various languages,

[5]Fourth Generation Languages.

in particular a *C* version is available. The host language independent part consists of several elements: a clause giving the global values needed for the event, a pre-condition expressing when the event can occur, output and input parts. The host language part, simply describes what updates to global values are possible. The globals used and updated are made explicit and thus tracing the effects of events is easier (although I would personally prefer even less reliance on global state). It would of course be possible to use other specification notations such as an algebraic notation or Z as the host language, but of course then the resulting system would not be executable.

SPI has a prototyping tool for use when only the eventCSP dialogue description has been produced. This allows the designer to examine possible event traces. Later full prototypes using the me-too version of eventISL or the C version can be executed. One drawback with the implementations that I know of is that they do not offer the full parallelism of the CSP. This is because the underlying languages they were built upon did not allow full multiplexed, non-blocking I/O. They fake the nondeterminism as long as they can for internal events, but when one of several choices of user input device are possible, the system makes an arbitrary choice. Most versions of C on UNIX or PCs have system calls for non-blocking I/O, so it should be possible to rectify this, at the cost of some loss of portability.

1.7 Non-committed prototyping notations

In the same way that general specification languages can be used for interface specification, general programming languages may be used for interface prototyping.

Probably the most well known prototyping language for interfaces is *Smalltalk* [37]. Partly because of its innovative programming interface, and partly because of their natural well suitedness, *object oriented systems* have been used extensively for interface design. Smalltalk has a paradigm *MVC* (model view controller) which is used for most of its programming tools (browsers, debuggers, etc.) and which is used widely by Smalltalk applications. The model is basically the object of interest, the view is how it is presented, and the controller handles input. The deficiencies of the paradigm are well documented, in particular the separation of input and output, which sounds sensible is often impossible in highly graphical interfaces, leading to complex dialogues between the parts, repetition etc. Other paradigms have been built upon Smalltalk, and it remains one of the most responsive (but easy to hang yourself with) prototyping environments available.

Prolog is often used for *knowledge based systems*, and thus it has been used for interface prototyping and production systems. Some of the "nice" features of Prolog for prototyping, in particular nondeterministic backtracking searches, can be a liability for interfaces. (Backtracking output is generally not possible.) Also the Prolog I/O primitives are in keeping with the procedural/imperative reading of the Prolog rather than its declarative, logical reading. On the other hand, several implementations of Prolog offer extensive support for windowing, menus, graphics etc, or have foreign function call facilities which allow graphical facilities to be easily added. Also there has been work on declarative graphics definition [99].

Alternative logics have been proposed that may be more useful for interactive systems specification. *Temporal logics* include notions such as "always", "until", "sometimes" which can be used to specify dialogues, again there are problems with backtracking, and specifications must be analysed for determinacy in order to be effectively executed. (This does not reduce the logics power for specification) In addition dynamic logic [103] can be employed which uses (in effect) actions to measure time flow. This has statements of the form `<A>P`, read "after the action `A` the proposition `P` will be true". Actions may be single user interactions, or composite (syntactic units).

Some *LISP* systems have extensive graphical support, especially *InterLisp*, which has a full multi-windowed, mouse driven interface. Like Smalltalk, it has supported many innovative interfaces and comes from the same stable, Xerox PARC. For example, the *hyper-text* system *NoteCards* [43] is built upon InterLisp. Like Smalltalk, the users of such systems are frequently dumped in the LISP debugger, and I think that most if not all systems developed are re-implemented in a conventional programming language. (This is probably a good thing as it encourages "throw away" prototyping or *rapid prototyping.*) LISP systems are also the host for various design tools (especially those developed from the US *AI* tradition) – for instance, *Trillium* [53], a frames based prototyping tool. However, in this too, one may be expected in such tools to drop into LISP to produce sophisticated effects.

Pure *functional programming* has already been mentioned above in relation to Cook's specification work [18]. He used a functional language derived from Cardelli, that has features with an object oriented feel. As far as I know this was never used for prototyping, and in fact Cook's interest has moved on to Smalltalk. Alexander [1] used techniques for dialogue design based on eager functional programming with an stimulus response model of interaction (called *ECS*). She also moved on to SPI, partly because of the difficulty of expressing parallelism. Although the implementation of SPI was itself in an eager functional language. On the other hand, in the specification exercise described above at York, we found we had problems whenever we departed from a pure style. In particular, impurities curtailed drastically our ability to reconfigure the system. We, in a similar fashion to ECS, used an event response paradigm. The top level of the specification was a state transition function

$$doit : State * Input -> State * Output$$

The function is applied to the current state and the current input event to produce the output and the next state.

With the possible exception that purists would replace the state by a higher- order function, this seems the only sensible I/O model for pure, eager functional programs. It is of course, however pure the language the state transition function is written in, a highly imperative paradigm.

Lazy functional languages have an additional I/O mechanism, the stream. The interaction is modelled as a single function with:

$$Output\_stream = f(Input\_stream)$$

Recent advances in compilation techniques mean that such languages are no longer as sluggish as they once appeared. *LML* for instance, in some well suited tasks, rivals Pascal implementations (that may well reflect upon Pascal implementations). In addition, any lack of speed may well be made up for with the reduced development time due to facilities such as polymorphic type-checking, complex structured objects, garbage collection and higher-order functions. The simple single input / single output paradigm make for highly constrained dialogues; for more rich structures the nondeterministic merge operator can be used which serves a similar role to, for instance, ADA select. Another problem lazy languages have is that it is often difficult to work out how exactly the input and output are interleaved, if the programmer is careless, the program may give half its response before the users give their input. In efforts to control this sort of behaviour interactive programs can contain hideous clauses like:

```
if ( x = x ) then ...
```

In addition, many standard transformation rules do not preserve the interactive semantics, hence special rules, or guards on standard rules have had to be developed [105]. In a similar fashion to specification notations, various higher order functions, or generating language have been proposed to encapsulate the interactive behaviour, effectively producing a dialogue sub-language.

Hyper-card is widely available on Apple MACs, and as well as being an interactive application in its own right, has been used as a development and prototyping tool for other applications. At the simplest level, it can be used as a simple slide show of application screens. More sophisticated prototyping can be achieved by using Hyper-card buttons to navigate around fixed screens in a way which mimics the actual dialogue structure. Finally the Hyper-talk command script language with low level language calls if necessary, can be used to prototype a large range of applications in full.

1.8 Dedicated interface prototyping and development tools

Some of these have already been mentioned above, EPROL and SPI for example. Tools may lay emphasis on the visual presentation or on the dialogue or both. The simplest presentation oriented tools are the screen painters which are too numerous to mention and packaged with most 4GLs.

The Alvey *Aspect project* has lead to the production of a very flexible graphical presentation manager called *Presenter* [132]. This allows arbitrary hierarchies of graphical regions, text and bitmaps with optional user controls for constrained movement, resizing etc. Graphical devices such as sliders, buttons etc. can be easily created and then reused in different applications. Because these are created rather than primitive, the designer is able to create new devices for a particular application or application family which are on a similar footing. The designer is thus not constrained by a set of predefined atomic devices. The tool is currently available with a *C* language interface, however it also has a graphical editor *DoubleView* [56] which enables the creation and modification of Presenter trees. The style of binding between application and Presenter allows easy reconfiguration of the interface presentation without recompilation or redesign of the application. Presenter deliberately restricts itself to presentation issues and does not deal with dialogue, this is regarded as a separate component. It does however allow quite a lot of dynamism, both under user and programmer control, and is thus far more than a screen format description. Typically an application will have a single Presenter tree describing the entire interface capability, time varying screens being obtained by dynamically changing parameters of the tree and popping or hiding regions. Single time multiplexed, multiple overlapping windows, tiled screens are all possible, as is the complete run time creation of screens. (The editor DoubleView is itself a Presenter application).

England's state transition dialogue tool [31] (described above) links into a screen formatting language (FDL) for generating interfaces with buttons, menus etc. An editor is also supplied for the graphical construction of such screens, which are then converted into the definition language. This description can then be used for the generation of applications. The possible screen formats are far more restricted than Presenter, being essentially static screens, but enforce a certain level of consistency within Eclipse tools.

Both Presenter and FDL are essentially for the description of the visual presentation of an application and do not attempt to address the issues of dialogue specification. As most dialogue description techniques tend to abstract away from concrete presentation issues they need to be used in conjunction with some presentation tool. Either of these tools could be used in this capacity to obtain a rather cleaner design than would be the case if a standard window manager was used directly.

PAC (presentation, abstraction and control [19]) is an object oriented dialogue design tool developed at University of Grenoble. It differs from the MVC paradigm in that input and output are regarded as aspects of the Presentation, recognising their heavy inter-relation. The Abstraction (like the MVC model) is the underlying object of interest and the Control maintains the consistency of the two. The control component is very important as it is assumed to have semantic knowledge about the abstraction, perhaps be able to supply context sensitive help. The control performs the job described as "linkage" by Cockton [15]. The model is hierarchical, each PAC object being composed of further PAC objects performing simpler tasks. It has been used to produce mouse based interfaces and is available on the MAC as well as other workstations.

Input-tools [136] is an example of a *non*-object oriented notation which translates into *C*. It was developed initially with interactive systems in mind, but later generalised to include process communication for *real-time* systems (cf. the movement in the opposite direction from CSP to SPI). Each tool description includes (like PAC) both input parts and output of prompts and echos. Each non-primitive tool is composed hierarchically of other sub-tools, the relation of sub-tool to parent tool being determined by a *regular expression* like syntax, e.g.:

```
tool  number (int result) =
input ( digit*  +  sign;digit;digit* );
...
```

The regular expressions thus form the "grammar" for the interaction. The tools also have value part (e.g., the integer result above), which can be thought of as their semantic component. This value part is far less clean than the grammar. For example, in the tool for reading numbers (taken from Plasmeijer [101]) the `digit` tool imperatively adds its digit into a buffer rather than being able to return it as a result to the invoking tool. This is because of problems with determining how to handle results of the *Kleene star*[6] operator (and choice). This messiness stands in contrast to the declarative feel of the regular expression. The output side is similarly messy, relying on explicit `prompt` or `echo` calls in the code part of the tools. This is another example where output expressiveness takes second place.

DICE [74] has a very similar feel to Input-tools. It too uses a regular expression like syntax to express the grammar part of the interaction, but has developed the value part somewhat further. So, for example, it could handle the Kleene star operator better, returning results from (its equivalent of) the `digit`, and confining the messiness to within the `number` dialogue cell. It is still ugly however, involving interleaving of parent cell computation with sub-cell activation, however this seems to be necessary within the paradigm as the sub-computations are also used to change factors such as echoing style.

DICE distinguishes between the semantic actions that produce the resulting value of the cell and those concerning feedback (`VALUE` and `ECHO`) this goes some way towards a cleaner output model but the examples still seem incredibly arcane.

Another example of problems with the value part of DICE is that it is hard to distinguish the values returned from multiple invocations of the same sub-cell within a regular expression. In the manual [74], an example is given of `mk_arc` which takes three user specified points and produces a circular arc. the regular expression part is given as `*(loc1)`. Strictly this is any number of

[6] Star closure: for any formal language L, the language

$$L^* \text{ is defined by } \{\Lambda\} \cup L \cup LL \cup LLL \cup \ldots$$

where Λ is the empty word.

points, but the actions determine that the repetition stops at three. (Why the example could not have been `;(loc1,loc1,loc1)` is unclear.) With either representation `mk_arc` needs to keep a counter to tell it which of the three locator values it is dealing with.

Even more ugly is the way the Kleene star operator is terminated. In Input-tools it is terminated when the sub-tool no longer matches (the match being determined by matches of lower level tools or by boolean guards), DICE instead signals this by assignment to a pseudo-variable (`@9 := 1`). A similar method is used to denote which of several sub-cells to execute in the `case` form of the regular expression part. Here the programmer sets a pseudo-variable `@8` to indicate which sub-cell to activate.

This failure to deal cleanly with the semantics of dialogue is common to all the prototyping techniques mentioned. From the point of view of final results of dialogues, a denotational approach (as advocated by Anderson [6]) would seem ideal, but this does not deal adequately with intermediate echoing.

A more specific worry I have about DICE, is that the description of the control structure seems to imply that the DICE cells will be called as the leaves of the program control tree [74], that is only the micro-dialogue is described. It is thus a way of creating a tool box of interaction objects rather than intended for specifying the entire user dialogue. The danger of such an approach is that it will lead to system controlled dialogue at the large scale, with the resulting problems of dialogue *over-determination* [130]. It is perhaps a realistic view of the way people write programs.

1.9 UIMS and Window managers

Whole books have been written contrasting different *UIMS* (User Interface Management Systems) approaches (e.g., Pfaff [100]). One of the original aims of UIMS was to separate applications from interfaces, allowing multiple interfaces to single applications and consistent interfaces within application families. They also promise dialogue prototyping with skeleton applications and various design aids. This worked fairly well for command based interfaces but have been struggling as graphical interfaces required more semantic feedback and the applications more display sensitive input. UIMS are not regarded as the panacea today as they were a few years ago.

The term can cover relatively clean and conceptually simple architectures such as PAC (which wisely does not use the term of itself), but also monolithic entities such as *GWUIMS* [116] (George Washington UIMS). This has lexical, syntactic and semantic (application) levels with interaction objects at each level and some bridging the gaps!

Various dialogue specification formalisms have been used within UIMS. State transition networks are common with various prefixes: Augmented TNs, Generalised TNs, Generative TNs. Expressing parallelism and multiple instances of the same dialogues within simple state transition networks lead to exponential explosions in the numbers of states. This is one of the primary reasons for some of the more complex extensions. Other problems include the proper treatment of output and errors.

Production rules (similar to expert system techniques) have been used also, these have a natural parallelism but have the dual problem of expressing sequence succinctly. The dialogue designer is likely to end up reinventing program counters!

Took's thesis [133] will contain an exhaustive review of UIMS and graphical display notations and packages. A pertinent section of this is included in Appendix A.

Window managers do not usually attempt to address dialogue issues. Normally they supply a toolkit of interaction objects such as menus and windows, frequently leaving the designer to perform many functions at the level of bitmaps. There is typically little evidence of a clear conceptual model; one is instead left with a rather muddled view of the hierarchy of implementation objects (e.g., panels, windows, sub-windows) with different operations at different levels. This is one of the problems that Took explicitly attempts to address with Presenter's simpler, but more flexible conceptual model.

One of the issues that has arisen particularly in UIMS and window managers is the issue of internal versus external control. Basically if the environment supplies services that the programmer can call to perform user interface functions then it is said to have *internal control.* On the other hand, if the environment takes control asking the application to perform services when it requires, then this is called *external control.*

Where (as may be the case with a UIMS) there is separate dialogue description component that is exercising the external control, this seems a good thing from the user interface point of view, as it is easier to trace the flow of dialogue (see Section 4.2.1 for a discussion of these issues). To counter that, the advantage of internal control is that the application may be able to make more informed dialogue choices than the UIMS. In particular, there is the issue of *semantic feedback* that is continuous responses that depend on application semantics. In principle the UIMS could continually ask the application for whatever semantic information is needed. However this has severe performance penalties.

With window managers there are different issues. The window manager will typically not include any dialogue control (except in the form of micro-dialogues like menu selection) and thus this is always coded in the application program. External control here refers more to the calling of service routines in the application when quite low level events (e.g., mouse clicks) occur. External control tends to foster user controlled dialogues and to encourage the programmer to put all the dialogue code together, but the form of the code will become highly convoluted. This also will be discussed further in Section 4.2.1.

Overall, the general feeling one gets from this field is

"Never mind the formality – feel the bits."

Chapter 2

Abstract mathematical models of interactive systems

The basic idea of developing abstract mathematical models and using them to specify useful properties of interaction has been developed in various ways at York. Many of the models are described in Dix's thesis [23]. The more recent work on templates and cycles is reported in [47].

2.1 Simple PIE model

The first model studied at York was the *PIE model*, and its close derivative the *red-PIE*:

$$\begin{tikzcd} & & D \\ P \arrow[r, "I"] & E \arrow[ur, "d"] \arrow[dr, "r"'] & \\ & & R \end{tikzcd}$$

Figure 2.1: The red-PIE model

Roughly, P represents the user's inputs, however it can be used at either the lexical level of keystrokes, mouse clicks etc., or at a more syntactic or semantic level of the design. E, the effect, depends again on interpretation and level of abstraction. Sometimes, only certain features of the system's state and output may be considered important, and others the entire *observable* state is required. The d, or display mapping yields the part of the effect which is immediately available, and the r or result part the "final" effect, usually closely related to the user's goals and task. In, for example, a word processor, the display might be the actual screen and the result, the printed document.

Clearly, the level of abstraction at which the model is applied strongly influences what principles are important, and a major part of the designer's role when using these models is choosing this level. More about the role of abstraction in the display and result, and how varying them can include features of user knowledge in Section 2.4.

Even this simple model can enable us to discuss issues such as:

Observability: Can I view the entire result via the display?

Reachability: Can I do anything I want to?

The PIE model is discussed further in Chapter 3 in the context of a Z representation of it and the PRG model. The rest of this chapter will deal with some of the other more domain specific and specialised models studied at York.

2.2 Derivatives of the simple models

In order to study more specific areas of interactive behaviour, various refinements and derivative models have been used.

2.2.1 Windowed systems

These are described in [23] and [25]. Here we have been interested in the problem of interference between windows. Where different windows represent different tasks, the user wants to treat these as independent. That is, while there is only one user there are effectively several different personae, one for each task. Hence a multi-windowed system is rather like a multi-user system, except without most of the protection mechanisms that operate between users. Interference between windows can therefore be very damaging and understanding and defining forms of sharing between windows is crucial.

2.2.2 Temporal models

Real-time interface issues, such as keyboard buffering and display strategies have been considered. Mechanisms to ensure predictability even when the system response time is slow have been specified. See [24] or [23]. One of the most important results of this work has been the realisation that often the low level services provided by operating systems and window managers make it *impossible* to produce usable applications.

2.2.3 Nondeterminism

When considering various formal properties of the models above there arose the need to use nondeterministic models. For instance, in describing the interaction with one window ignoring the interleaved actions in other windows, the perceived effects will be nondeterministic if there is any interference. Considering this *formal* nondeterminism led to an *informal* recognition of nondeterminism as a real interface phenomenon, and prompted the analysis of many common interface problems in terms of the paradigm [23].

2.2.4 Editing the view

Often interaction is modelled linearly, the user's commands affect the system state, from which display feedback is generated. We have also considered models where the focus of attention is on the display, and the user's commands are seen as affecting the display directly, the internal state being changed in accordance. This leads to a different way of understanding the interface, in particular it focuses the designer's attention on what should remain *constant*. For instance, if designing a new component on a CAD system, and the system runs out of room, it would be perfectly consistent with our view (the new component) to delete existing components in order to make room. This is clearly not acceptable, and no one would be likely to design such a system. However as the data base and the views of it become more complex it is not so easy to know what is or is not acceptable. Hence, for any view the user has of the system we must specify a complementary view that remains unchanged [23, 45].

2.3 Refinement and constructive models

The major aim of the above models has been to *define* useful properties. There is then the issue of actually building systems that satisfy them.

2.3.1 Conflict between interactive systems design and formal refinement

Having produced specifications that satisfy desirable properties, it was found that the specification structures that were designed to match the user model were, of course, very ill suited to implementation. This conflict will arise with *any* specification of interactive systems. If one leans towards efficient implementation structures, then it is likely that user requirements are badly defined, but if you lean towards the user then inefficient structures result. The necessary *structural transformation* required between the two poles itself conflicts with good software engineering practice in terms of modularity.

The problem is described in [26], as is one method of helping to solve the structural transformation problem *interface drift*.

2.3.2 Dynamic pointers

Manipulative operations are of two major forms, content based and indicative. The former includes operations of the form "colour all small triangles green" where a description is used to provide context. "Delete *this* word" is typical of the latter, where the subject is described indicatively. Mouse based systems have an obvious preference for the indicative mode of operation. In addition, such systems have highly display sensitive input, the meaning of "this" is the thing displayed on the screen. Clearly in developing a formal understanding of such systems, we need to analyse the meaning and properties of mappings between screen positions and what they denote.

Further, both in command based dialogues and mouse based systems, the "context" of the interaction can be classified similarly into content (the current search string) and position (which part of the folder is on screen). During manipulations on the underlying objects this context must retain its *semantic* identity.

Dynamic pointers are an abstract formal concept, capturing the notions of semantic integrity through mappings between interface level and changes in data objects. The features described by dynamic pointers are (necessarily) found in parts of most systems, yet often implemented inconsistently or badly. Focusing on this concept as part of the design process (and possibly also implementation) will help prevent such inconsistencies. In addition, it eases otherwise complex features, such as incremental parsing and multiple views.[23]

2.4 Bringing user models into the system model – templates

One of the problems with system models is that although they describe interactive behaviour they have no conception of how the user sees the system. We demonstrate what we mean by an example from [8] in the context of the predictability principle. One way [27] of making a principle of predictability precise is to require that if the effects of any two keystroke sequences are the same then no future experimentation will betray any difference in effect between the systems. This principle may be expressed formally as follows:

$$\forall\, p_1, p_2 \in P : I(p_1) = I(p_2) \Rightarrow \forall\, p \in P : I(p_1 p) = I(p_2 p)$$

For all p_1 and p_2 members of P, if the interpretation of p_1 equals the interpretation of p_2 then for all further $p \in P$, the interpretation of $p_1 p$ is equal to the interpretation of $p_2 p$.

This notion stresses that the *effect* is sufficient to determine the equivalence of distinct system states. From a user's point of view the fact that the two effects are identical may not be sufficient. It is at this stage that the designer *offers a hand* to the user modeller. How does the user explore the whole effect in order to ascertain what it contains? A stronger requirement of predictability (which as it happens may be too strong for any realistically complicated interactive system) is that the equivalence of the effects will depend on the *user's perception* of whether the effects are equivalent. If we regard the display as what the user perceives then it may be more appropriate to define predictability as:

$$\forall\, p_1, p_2 \in P : d(I(p_1)) = d(I(p_2)) \Rightarrow \forall\, p \in P : I(p_1 p) = I(p_2 p)$$

We can go further than this and add structure to the model to incorporate claims about user *attention*. Certain components of the display are more likely to be noticed in making decisions about the next action than others. Some parts of what is seen of the system will be different in a way that is irrelevant to the future of the application (for example more general system status information and the time and date). A claim about the design of an interactive system may be formulated in terms of a notion of a display template.

$template_D : E \to D$ extracts from the display that part which is claimed to be significant as far as the user is concerned. Hence predictability may be further refined into a more "attention" related property:

$$\forall\, p_1, p_2 \in P : template_D(I(p_1)) = template_D(I(p_2)) \Rightarrow \forall\, p \in P : I(p_1 p) = I(p_2 p)$$

To summarise the point: as far as the designer and the implementation is concerned Display is significant. The template, $template_D : E \to D$ adds structure to the designer's model that is not required in the implementation but encapsulates a claim about usability of an interactive system that may subsequently be tested. The system model, with user orientated structures, reflects a theory that must be tested. One of the roles of evaluation is to test mismatches between the theory and the reality of the implementation.

2.5 Making structure usable – cycles

This incorporation of additional structure into the system model, although it is not required for system implementation, may be particularly valuable as a means of clarifying user difficulties and as a framework for understanding the system. System modelling may be used to develop understanding of the structure or concepts of the interactive system. This structure will help the designer to conceptualise the important features of the design and may also produce a formulation that is relevant from the user point of view. Inevitably this user-orientated superstructure should map closely to the user task.

This idea of "superstructure" may be illustrated as follows. A structure that is particularly common in menu-based systems is what we shall call a *cycle*. For example a "main menu" presents an initial set of options. Each option leads to a dialogue sequence. The sequence completes some action (indicated by a change in the result) only at the end of the sequence when the main menu is redisplayed. Hence in a reference database [47], a *select* entry in the main menu will give way to a cycle that results in the selection of a reference given a particular name or date or source. This structure may be formalised within our PIE model in order to make the notion precise. The important thing however is that this notion of cycle is especially useful to the designer if it is a structure that is understood and used by the user of the system. It then becomes important to ensure that the relevant display at the start and end is clearly significant to the user and that whatever effect the system has on the result takes place at the end.

Using the formalism of the previous subsection, we may say that a cycle is a sequence of keystrokes that is viewed in the context of a *display template* that extracts the significant start and end display, and a *result template* that extracts the part of the result that is to be crucially affected by the dialogue sequence. The sequence begins when the display exhibits certain properties and completes at the earliest stage at which the same properties are exhibited:

$c \in P$ is a *cycle* with respect to a context $p \in P$, a display template $template_D$ and a result template $template_R$:

$$cycle(c, p)\langle template_D, template_R\rangle \Leftrightarrow$$

$$\begin{aligned} &\forall\, a, b \in P \,\text{s.t.}\, (a\, b = c) \wedge (a \neq nil) \wedge (b \neq nil) \\ &\quad \wedge (template_D(I(p\, c)) = template_D(I(p))) && [1] \\ &\quad \wedge (template_D(I(p\, a)) \neq template_D(I(p))) && [2] \\ &\quad \wedge (template_R(I(p\, a)) = template_R(I(p))) && [3] \end{aligned}$$

The three clauses require (1) that the first and last displays contain the same template information; (2) that the template display does not reappear during the cycle; (3) that there is no update to the result during the cycle. An *effective* cycle may also modify *result template* information but only at the end of the cycle.

$$effective\ cycle(c, p) \Leftrightarrow cycle(c, p) \wedge (\, template_R(\, I(p)\,) \neq template_R(\, I(p\, c)\,)\,)$$

A cycle may in fact be a complicated structure of subcycles; it is not our purpose to deal with minutiae here [47]. This notion of cycle is, in some circumstances, a mirror of the task notion as it would apply in the user model. With the aid of this hypothetical structure we may make claims about the design based on the concept itself and other related properties.

- *asynchronicity*: there may be cyclic structures that prematurely update the result, or apparently update the result misleadingly because the image of the result is updated on the screen [46];
- *template ambiguity*: where no unique display template information is discernible. The user fails to register that the cycle has properly terminated;
- *context sensitivity*: where a sequence of keystrokes may or may not be a cycle depending on the keystroke sequence that has taken place already.

This section has considered the interface mapping from one perspective – that of the system. A system model is useful for describing, in formal terms that are uncluttered, general properties of a sequence of system states. Notions such as *predictability* and *cycles* are high level abstractions over relationships between system states. These primarily serve the purpose of constraining and guiding the process of implementing an interface but they also relate to what the user does. Hence, those abstractions potentially provide a mechanism for directly formulating claims about usability. As we have seen, in order to formulate those claims in a manner that is sensitive to the presence of the user, structure needs to be added to any basic system model.

2.6 Limitations and application of abstract models

The biggest danger of any formal approach is that the designer may attach more credence to it than it deserves. This is particularly true of abstract models of the sort described here. An unwary user of the techniques may believe that formal consistency with some of the principles and models described above was *sufficient* for a system to be usable. The fallacy of this is obvious to anyone with an understanding of HCI[1], but formalisms can be both seductive and blinding. It is therefore essential to understand the limitations of the techniques.

Rather than being sufficient conditions for usability, the various formal statements of principles tend to be *necessary* conditions. So for instance, the various information oriented representations of the "what you see is what you get" slogan in [23] and Section 3.4.3 say, in forms applicable to different systems, that it is possible to work out the end result of a system by interactively examining it. This is clearly essential if the system is to be usable at all, but does not tell us how easy it is to work out the result, or how visually and spatially faithful a representation of the result we see on the screen. The latter of these problem is perhaps easy for the designer to verify, and mistakes will be obvious, but the former requires a deep understanding of human cognition that is unlikely to be formalisable to the same extent.

Some of the psychological issues are just beyond the scope of the models, and one can simply note that even when the system has passed the formal tests, more human centred analysis must be applied. In other places the two approaches marry together. Section 2.4 exemplifies this, the *templates* introduced there are intended to capture what the user notices of the display at any particular point in the interaction. In order to satisfy the formal principle the designer would have to give the relevant template functions. These can then be validated either empirically or by a human factors specialist. Similarly, the *strategies* by which the user can investigate the system state [23] are a form of user program, and can thus be validated against executable cognitive models, or again by direct evaluation. Often the simple fact that the operations and deductions that the user must perform have been explicitly stated as part of the formal proof will be sufficient to see whether they are reasonable or not.

The other major non-formalisable part of the use of these models is deciding which principles are applicable and desirable to a particular application, and also at what degree of abstraction to apply them. A system may be understood at various levels of abstraction, concrete keystrokes and mouse actions, syntactic units, semantic commands. the models can often be applied to the system at each level. Some of the properties will be universally applicable to all systems at all levels, but in general the designer will be more selective. So for instance, in a command based operating system, one expects to have a total view of the current command being edited, that is it obeys a very strong visibility principle. However, when the command is submitted (entering the carriage return key) the results of it on the file system will be far less visible, usually requiring explicit commands to view files, directories etc. Arguably, in this example, the semantic level

[1]Human-computer interaction.

could do with being more visible too, but it is a design decision as to what degree of visibility is required at which level.

In short, as with any method or model the domain of applicability of abstract models must be born in mind when they are used or evaluated.

Chapter 3

Modelling in Z

3.1 Formal methods and HCI

In the past ten years, two separate movements have come to the foreground of computer science – *formal methods* and *human-computer interaction*. Formal methods exploit the unambiguous rigour of mathematics to describe the properties of a computer system. A formal approach to system design allows the investigation of properties of an information system without having to worry about implementation details. Reasoning at an abstract level of detail is often simpler and can prevent the manifestation of poor design decisions in live implementations. Human-computer interaction (*HCI*) deals with the relationship between an interactive system and its user. It is generally recognised that in order to produce high quality software attention needs to be focussed not only on the application task but also on the interface between the underlying information system and the user.

Formal methods have been successfully applied to the underlying information system. This chapter will discuss attempts to apply formal methods techniques to the interface. In particular, it presents the *PIE model* [28] rendered in the Z specification notation and then describes the model developed from it at the PRG in Oxford [125] which offers a bridge between the very abstract models like the PIE model and the practically-oriented methods such as formal grammars and state transition diagrams.

3.2 The problem

An *interactive system* consists of two essential features – an *application program* and a *user*. A *conversation* between the human and the computer consists of interaction *events* – either the human issuing *commands* as input to the computer or the computer producing an end product or intermediate display for the user's benefit. Any formal discussion of an interactive system will describe this conversation.

What are the interesting questions one can ask about the *human-computer interface* for an interactive system? A few of these might be:

- Is the interactive system both easy enough for novice users and powerful enough for experienced users?
- Is it clear to the user what changes to the underlying information structure result from commands that she issues?

- Do the intermediate views of the information state help the user predict the subsequent result of future commands?
- Can any concrete meaning be attached to the slogans popularised by promoters of various interactive systems (such as *WYSIWYG* [1] or *direct manipulation*)?
- Are there any common features of one interactive system that can be applied to new interactive systems, and can these features be captured in an overall model of interactive systems?

The answer to the first question is out of the scope of this chapter, for it wanders into the psychological realm of HCI, touching upon such issues as knowledge representation and user-modelling. It is in attempting to answer the last question that we answer all of the remaining questions.

Part of the answer to the problems of HCI is in properly formulating questions as above in such a way that their answers can be 'calculated' by some already familiar formal methods. It matters not so much the actual language of specification that one chooses as long as the language is sufficiently rich to express the questions.

3.3 Abstract mathematical models

Many of the methods discussed have been very concrete in order to describe and reason about specific interactive systems. This is a necessary consequence of the ultimate goal of computing science – the creation of workable systems. Work of a more abstract nature – not intended to lead directly to implementations of a particular application but rather to provide guidelines for future implementation attempts – has slowly gained momentum in the past five years. The Human Computer Interaction Group at York University has produced much of the literature concerned with the more general and abstract features of interactive systems. Motivation has been given for general mathematical models that can be applied to all interactive systems [49, 131]. Thimbleby proposes the use of generative user-engineering principles as a means of connecting the world of informal slogans to the world of formal mathematical descriptions.

In [28] and above (Chapter 2) a very simple abstract model, the PIE is introduced. The PIE model assumes that a sequence of commands form programs (denoted by the set P) which are then interpreted by a function (denoted by I) to their respective effects (in, of course, the set E). Figure 3.1 is a pictorial representation of the PIE model. Even at this seemingly naive level many properties of interactive systems have been discussed for the first time in the unambiguous style of mathematics. One of the simplest examples can be seen in the "gone for a cup of tea" problem. The user has been away from the computer (presumably in search of the elusive cup of tea) and may have forgotten what she was doing before she left. We are concerned with whether she will be able to predict the consequences of her next command based upon what appears at the screen. The real question in this example is whether the interpretation function I is ambiguous, i.e., do different command sequences produce similar immediate effects but display dissimilar future effects?

It is desirable at this point to cease with the strictly prosaic presentation of the abstract PIE model. But before we begin our brief formal tour, some explanation is needed. The presentation of the PIE model in [28] gives definitions like the one above for ambiguity with a simple mathematical notation that can be understood by anyone with a basic mathematical background. The reason

[1] "What You See Is What You Get."

$$P \xrightarrow{I} E$$

Figure 3.1: The simple PIE model

for casting these definitions in the language of Z [120, 121] is purely for a uniform presentation throughout this chapter. It is again important to note that the actual choice of notation is not that crucial as long as it preserves the explicative power of the language of mathematics. We are usually only dealing with simple relationships between sets of objects (called "*types*" in Z parlance). A fortunate convention of the Z language is its dependence on prose as well as mathematics. A good explanation in Z will consist of excerpts of formal notation separated by descriptive English prose. The purpose of the English (or any other indigenous language for that matter), is to aid the reader in attaining an intuitive grasp of the ideas being presented. The formal notation should then solidify the intuition with an unambiguous piece of mathematical text. Any failure of the presentation in this chapter to satisfy both intuitive and formal appetites should be attributed to the author's poor style, as opposed to any shortcoming of Z. What follows assumes no prior knowledge of the Z notation; however, it should not be considered sufficient as a tutorial introduction to Z. Interested readers are directed to [121].

We begin our formal presentation of the PIE model with the sets representing the programs and the effects, respectively, P and E. We are not concerned at this point with any internal structure of these two sets, just that they represent all of the possible programs and effects relevant to our discussion. To assert the existence of these two sets in Z so that we may refer to them throughout our discussion, we simply write

$$[P, E]$$

The PIE model can be represented by using a *schema*, a special type in the Z notation that corresponds roughly to an unordered tuple. A schema consists of a declaration of schema components (given with names and associated types) along with predicate constraints on the components. Since a schema is a Z type, it can be viewed as a set of objects. The elements in the set described by a schema type are precisely those bindings of the components which satisfy the predicate constraints. As an intuitive guide, it is not bad to think of a schema as corresponding roughly to a record in a language like Pascal, though this analogy should not persist with increasing familiarity of Z. We will present the schema type for the PIE model and then fully explain it.

PIE

$programs : \mathbb{P}\, P$

$effects : \mathbb{P}\, E$

$interpretation : P \mathrel{\rightarrow\!\!\!\!\!\!+\,\,} E$

$\mathrm{dom}\; interpretation = programs$

$\mathrm{ran}\; interpretation = effects$

$\forall\, p, q : P \bullet p \frown q \in programs \Rightarrow p \in programs$

The name of the schema type is given at the top (PIE). An object of type PIE will be one occurrence of a PIE. The first section of the schema – the *signature* – consists of the three components *programs*, *effects*, and *interpretation*. For a given object of type PIE, say $mypie : PIE$, we denote the components of *mypie* by *mypie.programs*, *mypie.effects*, and *mypie.interpretation*. The component *programs* represents some specific set of programs obtained from the set of all possible programs, P, introduced above. (Note: the colon used after the name of the component here, and at all times, can be directly translated to "of type". Hence, the first line of the schema can be read "*programs*, of type power set of P"). The notation $\mathbb{P}\, P$ (pronounced "power set of P") is a type derived from the type P and it represents a set all of whose elements are of type P. If the reader is confused, consider what we would mean if we had written $program : P$. This says that *program* is of type P and so it is any one of the possible programs we wish to consider. But *programs*, as defined above, is not just any *one* program. It is a whole bunch of programs. Similarly, *effects* is some set of specific effects obtained from the set of all possible effects, E. The final component, *interpretation*, is a mapping from programs to effects. (Technically speaking, the symbol $\pfun$ represents a partial function, but we need not worry about that now.) In the future, when we write that something is of type *PIE* we would mean that it has three components representing the programs, effects and interpretation mapping.

The bottom half of the schema – the *predicate* – presents the constraints on the components of the signature. The first constraint says that the domain of the interpretation mapping is exactly the set of programs in the PIE. A mapping takes elements from a source set to a destination set. The elements in the source set are what we call the domain. The elements in the destination set are what we call the range. The second constraint says that the range of the interpretation mapping is exactly the set of effects in the PIE. The last constraint simply says that given any two programs (taken from P) such that their combination is contained in the set *programs*, the first program is also contained in the set *programs*. We say the set *programs* is *prefix-closed* in this case.

Now that we know what it means when we say "let *mypie* be of type *PIE*", we can return to the discussion of the ambiguous interpretation function mentioned above (the "gone for a cup of tea" problem). Expressed mathematically as an axiom in Z we have

$$
\begin{array}{|l}
ambiguous : \mathbb{P}\, PIE \\
\hline
\forall\, pie : PIE \bullet pie \in ambiguous \Leftrightarrow \\
\quad (\exists\, p, q, r : pie.programs \bullet \\
\qquad pie.interpretation(p) = pie.interpretation(q) \land \\
\qquad pie.interpretation(p \frown r) \neq pie.interpretation(q \frown r)\,)
\end{array}
$$

The set *ambiguous* includes all 'ambiguous' PIE models. Given a PIE we can tell if it is ambiguous under the interpretation function I if there exists two programs, p and r, whose interpretations are identical (i.e., they yield the same effect, or $I(p) = I(q)$) but when extended by the same command sequence r, yield different effects (i.e., $I(p \frown r) \neq I(q \frown r)$). If $mypie : PIE$ is ambiguous, then $mypie \in ambiguous$ is *true*.

This simple PIE model goes a long way in describing many properties of all interactive systems. Dix and others have expanded the model in several ways; see [23] for an exhaustive discussion of PIE models. Runciman has even designed a rapid prototyper for the PIE model based on a functional programming language [107]. It was this PIE model that served as an initial inspiration for Sufrin's model discussed in detail below. The PIE model does not exactly represent a specification language for interactive design. The reason for its inclusion in this report is that its generality epitomises the kind of approach to interactive software development that must be followed. A notation is necessary in which properties of interactive systems can both be expressed easily and proved rigorously. The PIE model suggests how to phrase properties of interactive systems with mathematical precision. The language of mathematics provides the machinery for rigorous proofs. The Z notation provides a notation for more clearly stating the mathematics for an audience composed at least of systems designers and, hopefully, system users. A successful specification technique should able to address both needs.

Anderson has also provided an abstract model of interactive systems for the purpose of proving properties of general systems [6]. His model is more complicated than the PIE model and its presentation now would not offer any new insights.

The abstract models presented in this section adequately describe general properties of interactive systems. They serve as an inspiration for a more constructive specification technique developed at Oxford, which we shall now explore in detail.

3.4 The PRG model

In this section, we present the Z-based model of interactive processes as developed at Oxford University's Programming Research Group. We have produced a mathematical model of an interactive process by using a hybrid approach which permits "*state* oriented" and "*trace* oriented" styles of specification at once. A process evolves by participating in events, during which it undergoes a transition from one state to another in some nondeterministic way. This nondeterminism is important because its presence prevents overconstraining implementations and allows us to abstract away from the complete history of a system. Nondeterminism also supports the description of a system from one of a variety of perspectives.

3.4.1 Introduction to the model

We begin once again by introducing our basic sets. We will denote the set of all possible states in which a process might be by the set S, and the set of all possible events by the set E. For the sake of the model, we are not interested in any further detail associated with these two sets.

$$[S, E]$$

Given these sets, we can describe the components of a *process*. A process has an *alphabet* which is a set of events from the set E in which the process may participate. Each event has an associated *behaviour* which details the state-to-state transitions brought about by the occurrence of the event. By describing the behaviour function, we are implicitly indicating the allowable sequence of events for the process. The behaviour function is similar to the state transition diagram method mentioned earlier. In addition, we can make explicit the allowable sequence of events by specifying the *traces* of the process. A trace is similar to the programs mentioned in the PIE model, serving as a history of the events in which a process has participated. We can specify the traces as a formal grammar, and in this model we choose a notation similar to *CSP*. This hybrid approach – borrowing ideas from formal grammars and state transition diagrams – enhances both expressiveness and readability.

The final component of a process is the set of initial states from which the process is initiated. In Z, we can specify a schema called *PROCESS* as specified by giving the following four components:

- the *alphabet* – the set of events in which the process participates
- the *behaviour* – an alphabet-indexed family of state-state transitions
- the set of *traces* – each trace is an allowable sequence of events in which the process participates

- the set of *initial states*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PROCESS \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
alphabet : \mathbb{P}\, E \\
behaviour : E \rightarrow\!\!\!\!\!+ (S \leftrightarrow S) \\
traces : \mathbb{P}(\mathrm{seq}\, E) \\
init : \mathbb{P}\, S \\
\hline
alphabet = \mathrm{dom}\, behaviour \\
\forall\, s, t : \mathrm{seq}\, E \bullet s \frown t \in traces \Rightarrow s \in traces \\
\hline
\end{array}
$$

The first constraint says that we are only interested in the behaviour of elements in the alphabet of the process. The second constraint says that for any trace of the process (i.e., allowable sequence of events – a program), all initial subsequence of that trace is also a trace of the process.

The behaviour function above only describes the effect on the state transition as dictated by single events. It is easily extended to demonstrate the effect of a sequence of events. Using a notation similar to *CSP* [55], we can define a set of operators which construct sets of traces, making it more convenient for us to specify the traces of a process independently of the effect each event has on the process state.

The following Z specifications may be skipped by readers who are unfamiliar with the notation, but is included here for completeness since these definitions are used in subsequent specifications. See [125] for further details.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PROCESS \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
PROCESS \\
behaviour\_tot : E \longrightarrow (S \leftrightarrow S) \\
eff\_behaviour : E \longrightarrow (S \times \mathrm{seq}\, E \leftrightarrow S \times \mathrm{seq}\, E) \\
behaviour\_seq : (\mathrm{seq}\, E) \longrightarrow (S \leftrightarrow S) \\
\hline
\forall\, e : E \bullet \\
\qquad e \in \mathrm{dom}\, behaviour \Rightarrow behaviour\_tot(e) = behaviour(e) \land \\
\qquad e \notin \mathrm{dom}\, behaviour \Rightarrow behaviour\_tot(e) = \\
\forall\, e : E \bullet eff\_behaviour(e) = \\
\qquad \{\, s, s' : S;\, h, h' : traces \mid (s, s') \in behaviour\_tot(e) \land h' = h \frown \langle e \rangle \bullet \\
\qquad\qquad ((s, h), (s', h'))\, \} \\
\forall\, p : \mathrm{seq}\, E \bullet behaviour\_seq(p) = \\
\qquad \{\, s, s' : S \mid (\, \exists\, h, h' : traces \bullet ((s, h), (s', h')) \in {\scriptstyle ⨾}/(p \,⨾\, eff\_behaviour)\,)\, \} \\
\hline
\end{array}
$$

Now we are ready to define the *interactive process*. The interactive process is driven by a user prepared to participate in a subset of events called *commands*. As each command is accepted, the process moves autonomously from its current state through a succession of one or more states until it is ready to show the user some visible indication or view of its state, which it does "just after" engaging in one of the *show* events. In addition, the process is sometimes prepared to yield the user a result of some kind, which it does "just after" engaging in one of the *yield* events.

Here we are differentiating between a "finished product" and an interim interactive view of a system. An example of a finished product for a text editing system could be a printed document and an interim interactive view is the display at the screen at any given time during the editing session. The finished product comes from a set of all possible results, R, and the views come from a set of all possible views, V. Both the views and results are derived from the state and can be represented in Z as relations, or mappings, from the state space S to R (for results) or V (for views).

To turn a process specification into an interactive process specification we need to describe

- the events which the user can initiate (the *commands*).
- the *show* and *yield* events.
- the view (resp. result) which is seen after a *show* (resp. *yield*).

$[R, V]$

$$
\begin{array}{|l}
\hline
\textit{INTERACTIVE\_PROCESS} \\
\textit{command} : \mathbb{P}\, E \\
\textit{show} : \mathbb{P}\, E \\
\textit{yield} : \mathbb{P}\, E \\
\textit{view} : S \leftrightarrow V \\
\textit{result} : S \leftrightarrow R \\
\textit{PROCESS} \\
\hline
\forall\, t : \textit{traces};\ e : \textit{show} \mid t \frown \langle e \rangle \in \textit{traces} \bullet \\
\qquad (\, \textit{behaviour\_seq}(t \frown \langle e \rangle)\,)(\!| \textit{init} |\!) \subseteq \operatorname{dom} \textit{view} \\
\\
\forall\, t : \textit{traces};\ e : \textit{yield} \mid t \frown \langle e \rangle \in \textit{traces} \bullet \\
\qquad (\, \textit{behaviour\_seq}(t \frown \langle e \rangle)\,)(\!| \textit{init} |\!) \subseteq \operatorname{dom} \textit{result} \\
\hline
\end{array}
$$

(Note: the function *behaviour_seq* is simply the originally defined function *behaviour* extended to operate on sequences of events.)

By including the schema name *PROCESS* in the signature of the schema above, we automatically include into *INTERACTIVE_PROCESS* all of the components of *PROCESS*, along with their associated constraints. This points out the modularisation attainable in Z and greatly enhances its readability. The two added constraints just ensure that we can obtain a view or result of the process at times when we would want to (i.e., upon issuing a request for view or result after a legal sequence of input events).

We have now presented the main features of the model of an interactive process. The rest of our discussion will be of a more informal nature and will concentrate on how to apply this model in reasoning about properties of interactive systems.

3.4.2 Formalising some user-engineering principles

When users work with an interactive system, their only means of evoking behaviour are commands, and the only consequences of the commands which can be inspected are results and views. The form our predictions of the interactive system will take will be a description of the relationship between command sequences issued by a user, and the results and views to which they (may) give rise.

We can formalise this notion of programs causing results and views as axioms over our model. In so doing, we will be able to cast user-engineering principles – existing in the past as a host of well-known and ill-understood slogans – as precise mathematical descriptions whose truth can be rigorously proved (or disproved). We will spend some time discussing some of these user-engineering principles. The interested reader is referred to [125] for more detail.

The result of a system is *command determined* – or, simply, *deterministic* – if a sequence of commands which yields a result always yields the same result. This property seems to be very desirable; but consider, for a moment, a text editor with a command which inserts the current date in the document being constructed. The presence of this command means that the result is not command determined, for at different times it will produce a different result (i.e., a different time). This may not sound like a big problem, but strictly speaking, the inclusion of such a command would prevent the whole system from being command determined. If it is the only such command, however, then the system nearly has this desirable property. So rather than labelling the whole system deterministic or not, we would label subsets of its commands as deterministic.

A set of commands is *complete* if they can be used to generate all results. The commands which are present in every complete set of commands are called the *essential* commands. A command sequence which cannot be extended to cause a result is called *fruitless*.

So far we have only discussed results. Similar principles apply for the view as well.

3.4.3 Relating views and results

It is generally accepted that the view offered by a system should help the user determine the result which is under construction. A text editor that does not show the user any changes after deleting a character from the text will only lead the user into confusion. We can formulate a whole series of user-engineering principles based on the relationship between views and results.

Two command sequences are *result equivalent* if they cause the same set of results; they are *view equivalent* if they cause the same set of views. We can imagine two computers side-by-side executing the same text editing program. On one computer a command sequence is typed at the keyboard and the display reflects the view to the user. A different command sequence is typed at the keyboard of the other computer and its corresponding view is displayed. The two command sequences are view equivalent if the two views are the same.

Extending a pair of equivalent command sequences by a single command may make them inequivalent, but two equivalent command sequences which can be extended indefinitely without becoming inequivalent are called *indistinguishable*. In our example above, if the two views are the same and, furthermore, as long as identical command sequences are typed at the two keyboards thereafter, the view remains the same, then the initial command sequences are view indistinguishable.

The commonly used slogan "what you see is what you get" (*WYSIWYG*) is hard to achieve literally. A milder version of this we call *visual consistency*. If two command sequences in a visually consistent system are view indistinguishable, then they are result equivalent. If we return to our two identical text editors, we have typed in two different yet view-indistinguishable command sequences, p_1 and p_2. If we type the same command sequence, s, to both editors now the view will always be the same. In a visually consistent system we are assured that the results yielded after (and only just after) both p_1 and p_2 are also the same. The corresponding slogan is "what you can find out (through a view) determines what you have got now (a result yielded)."

A system is *strongly visually consistent* if its view indistinguishable command sequences are result indistinguishable as well. Again returning to our two identical text editors, we have typed in two different yet view-indistinguishable command sequences, p_1 and p_2. If we type the same command sequence, s, to both editors now the view will always be the same. In a strongly visually consistent system we are assured that the results yielded after both p_1 and p_2 are also the same and will continue to be the same after each command in s. The corresponding slogan is "what you can find out determines all that you are going to be able to get." We believe that this is what is meant by the more popular "what you see is what you get."

3.4.4 Refinement

We have briefly set forth some features of an abstract model of interactive processes. We finally direct our attention towards refinement from a process specification to(wards) an implementation. If a relation S is used to specify the behaviour of a program, then its domain (dom S) characterises those states from which the program will terminate if started. A relation R is said to *refine* the relation S if it terminates when started in any state from which S would terminate, and does so in a state permitted by S. An implementation is allowed to "do anything" when invoked in situations where the precondition of its specification is not satisfied (i.e., on states not in the domain of S).

Based on this idea of relational refinement, we can formulate a notion of refinement of interactive process specifications. We find that refinement respects many of the properties defined earlier. We would like to be able to refine processes by choosing machine-oriented representations for the abstract information structures used in our specifications. This form of refinement is usually known as *data refinement* and is explained in detail in [65] and [86].

The basic idea is to establish a correspondence between the two state spaces, then to check that the behaviour of the implementation adequately "simulates" the behaviour required by the specification under this correspondence. In the "classical" formulation of data refinement, the correspondence is given as a relation, called the *abstraction relation* between the state space of the implementation and that of the specification. We can show that this process of *downward simulation* is both sound and closed under composition.

Chapter 4

General properties and issues

4.1 Dimensions of evaluation

The different notations and methods have been designed, and are most effective in different contexts. No single notation can be useful for all contexts, and even within a single project different notations are likely to be useful for different phases and concerns. For instance, even in the simple situation of students' design exercises, Sharratt [111] used a combination of CLG for most of the top-down interface design, but substituted a dialogue prototyping tool RAPID [141] for CLG's final, lexical level. This allowed his students to have the advantages of more abstract analysis at earlier stages of design, but an executable prototype at the end.

This section describes some of the dimensions in which the various notations fit. This is to enable the reader to focus on what the notations is required *for*, and, if the requirement is over a wide spectrum, what sort of package of notations may be useful.

4.1.1 User/System/Interaction orientedness

Viewing an interactive system as a dialogue between user and system, notations may lay emphasis in three areas:

User What is going on inside the user's head, what metaphors and models are being used, what are the user's goals? This is clearly the area addressed by notations like GOMS and user programs (in fact largely the first two sections of Chapter 1).

System The internal workings of the systems. What is the designer's model of it and what are the concepts by which it should be understood. Under this heading lie the general specification techniques and general programming and prototyping languages

Interaction The actual trace of tokens that flow between user and system ignoring the thoughts of the former and semantics of the latter. Grammar based techniques and dialogue specifications roughly cover this area.

In practice, notations rarely fit entirely into one category or another. The GOMS type models can break down tasks right down to the keystroke level; system notations describe the input/output relationship. in addition, dialogue specifications are rarely semantics free; some (such as the eventISL part of SPI) include specific semantics, and others include it implicitly, for instance, by grammar rule names.

It is interesting that the tendency is for the "interaction" notations to specify predominantly the user $\rightarrow$ system dialogue but ignore the system responses.

It is also worth noting, that this "interaction" component is the only part of the ensemble considered by some as HCI, whereas the three together (especially the match between user and system semantics) are necessary to really understand the process. In principle, a behavioural model of the system or user can be obtained by looking at the communication alone, but this is probably insufficient for a proper understanding of the system.

4.1.2 Life cycle

At the earliest stages of software design, requirements elicitation, various forms of task analysis can be used, for example *TAKD* [62]. In general, these fall out of the range of this report but some of the notations stretch back to this stage. For instance, the task level of CLG would be likely to be specified as, or soon after, requirements are obtained.

Later, the general structure of the system may be decided, but not the concrete dialogue. At this point, the level of abstraction available in the notation is crucial. For instance, GOMS could be used with terminals corresponding to abstract concepts like "send the mail message", or the eventCSP part of SPI could be used, with events corresponding to a similar level of granularity.

Finally, as the concrete dialogue is specified, notations requiring specific lexical level description (such as TAG) can be brought into play, as can most prototyping tools. In addition, many of the user oriented techniques, although potentially usable earlier as a design drivers, are in fact used only as an analysis tool at this stage.

Of course, the picture is not quite as simple as this, as interface design is usually viewed as an iterative process. So, for instance, methods used for analysis of prototypes must be capable of mapping back into the higher level notations used earlier in the design cycle ready for the next iteration.

The abstract mathematical models do not fit very well into this picture either, as they may be used *before* the explicit requirements of a particular system are decided. At this stage, general properties required of the final system may be decided. These properties may refer to any of the stages (but particularly high and medium level specification) and are mapped onto the specific notations used as formal constraints.

4.2 Issues for dialogue description notations

4.2.1 Distributed and centralised dialogue description

If the dialogue is described by a pure grammar, with no semantic element, then it is easy to look at the dialogue syntax in isolation, understand and evaluate it. On the other extreme, if we take a typical interactive program, aspects of the dialogue will be distributed throughout the code, making it difficult to trace the course of a typical interaction.

A notation that wishes to describe the semantics of dialogue as well as its syntax can try to retain the advantages of a simple syntactic description, by separating the semantic and syntactic parts, allowing the dialogue designer to examine the dialogue syntax in isolation. This is a *centralised dialogue description*. Alexander's SPI is exemplary of this approach to the extent of having separate sublanguages for the two parts. This also demonstrates another advantage of this approach. The same form of the syntactic dialogue description may often be suitable both for high level analysis and automatic coding (or run time interpretation). The semantic description, on the other hand, is likely to have a different form when generated for specification/prototyping or for inclusion on a production system. Separating the two allows reuse of the dialogue description with different semantic parts reflecting differing uses of the specification.

Alternatively, the notation can choose to put associated parts of the syntax and semantics together. This has the advantage that parts of the interaction can be examined in detail allowing the evaluation of the syntax and semantics in tandem It also has advantages of abstraction, associated semantics and syntax can be packaged together. Its disadvantage is that, like the typical program, it has a *distributed dialogue description*. One has to examine diverse pieces of the specification in order to obtain an understanding of the large scale flow of the interaction. The procedural notations such as DICE and also most object oriented notations fall into this category.

The two approaches are not fundamentally incompatible, given a notation of the former type, it would be quite easy to separate out parts of the dialogue syntax and present them with the associated parts of the semantic description. Similarly, with some distributed notations it is possible to go through extracting the parts specifying the dialogue syntax and look at these together. For instance, with DICE (or Input-tools) one could extract all the `SYMBOL` (resp. `INPUT`) clauses which contain the regular expression like sub-tool syntax. These would then form the centralised dialogue for analysis.

4.2.2 Maximising syntactic description

I have said that such extraction is only possible with some notations. The reason why this is not always possible, and is not usually possible for general interactive programs, is itself an important issue. Usually it is possible to isolate the parts that are responsible for input and output (identifiable by `print`, `read`, etc.). However, how these fit together into a dialogue is masked by the surrounding code. In particular, what would be syntactic in a dedicated dialogue grammar description may be coded semantically. For instance, in eventCSP, one could write:

```
Text_editor  =      mouse_press --> set_selection
                []  key_press   --> add_char_to_text
```

In a programming language one might have:

```
ev = read_event();
if ( ev.type == EV_mouse_press )  set_selection(ev.pos);
else                              add_char_to_text(ev.char);
```

In the second version an analyser would have to recognise that the boolean expression `ev.type == EV_mouse_press` corresponded to a simple dialogue decision rather than a deep semantic decision in the application. Another example of this problem arises in the DICE tool `mk_arc` discussed in Chapter 1. Because this uses a counter in order to decide what feedback to give, it is impossible to decide by simple static analysis that one form is given for the first point specified, a different one for the second and yet another for the third. As I also noted at the time, the actual example doesn't even allow a static analysis tool to determine easily that exactly three points are required.

Even more problems may arise in production systems or window managers or UIMS with external control. In these systems, the most obvious form of dialogue is completely user-controlled. If the designer wishes to provide any control over the input syntax then a "program counter" must be explicitly included. So, for instance, if we were operating under a window manager that calls a user routine `process_event`, we might have the following code for text editor selection (in C):

```
enum { normal, selected }  mode;

process_event( event ev )
{
    switch ( ev.type ) {
        case  button_down:
            ...
            if ( in_text ( ev.pos ) ) {
                mode = selecting;
                mark_selection_start(ev.pos);
            }
            ...
        case  button_up:
            ...
            if ( in_text ( ev.pos ) && mode == selecting ) {
                 mode = normal;
                 mark_selection_end(ev.pos);
            }
            ...
        case  mouse_move:
            ...
            if ( mode == selecting ) {
                 extend_selection(ev.pos);
            }
            ...
```

Figure 4.1: Example Window Manager code

The dialogue for the selection is distributed widely over the event loop, and further it is only by keeping track of the `mode` variable that we can see that they are linked at all. The code is for a mythical but quite typical window manager.

In each case, the problem is that elements of the dialogue can be given a syntactic form or a semantic one. Obviously more complex elements of the dialogue will require complex computed decisions, but where possible, the more syntactic the dialogue description the more easily analysable it will be. This concept underlies much of data-base normalisation procedures which try to move the decision as to whether an update is acceptable from the semantic realm to the syntactic.

4.2.3 Parameterised and dynamic interleaved dialogue structure

If we consider many interfaces, the possible screen displays can be easily enumerated. The order in which such screens are produced, and the detailed contents of fields may differ, but the basic screen designs are finite. Other systems are more anarchic, especially multi-windowed interfaces where user interaction may dynamically cause the creation of new windows. Thus there is a clear difference between static and dynamic screen presentations.

A similar and related issue arises at the level of the dialogue structure. Some dialogues can be described by a finite set of dialogue states between which the user may move, whereas others are far more complex. Clearly, multi-windowed systems will have a correspondingly dynamic dialogue structure. Perhaps the dialogues within each window have a fairly static structure, but the number of such interleaved dialogues varies at run time.

Some notations only address the structurally static dialogues. For example, most grammars, state transition notations and SPI which has a static process structure. Thus such notations would not (without modification) allow the expression of general multi-windowed dialogues. However, many *WIMP*[1] based systems do not require this level of generality. In addition, many run time systems may only allow essentially static structures, for instance, prototypes programmed in HyperCard (except for very complex scripts) or under most forms based systems.

This issue of dynamic dialogue structure is often linked with that of parameterisation. An instantiation of a parameterised dialogue often involves the creation of new screen resources. For example, we could imagine extending eventCSP to allow

```
Multi_window_editor = new_name(name) -->
                         ( Edit_window(name) [] Multi_window_editor )
```

The instantiation of `Edit_window(name)` implies that a new window and dialogue within that window would be initiated.

Both parameterisation and dynamic dialogue structures have problems however. they both make it more difficult to analyse the dialogue. Thus static notation opts for a sparser (and fundamentally less expressive) domain of application but allow a far greater degree of automatic or manual manipulation.

Thus, if a notation allows parametrised or dynamic dialogues, it should also not encourage their use except where necessary. If there is a choice, the dialogue should be encoded using the more static forms of representation.

4.3 Trade-offs

Many of the features one would like of interface design notations are the subject of trade-off.

Analysis vs. expressiveness. Using completely formal notations like Z does not lead to easily analysed interfaces; asking questions about the interface becomes a an exercise in theorem proving. When notations try to encompass all possible dialogue styles they may hit a similar problem of complexity. Simple, finite, non-parameterised dialogue specifications are very easy to analyse. Questions, such as "Are there any dialogue situations from which you cannot recover?", become simple graph analysis problems such as connectivity and are thus amenable to standard automated solution.

User oriented vs. implementation. Formal notations are easily manipulable, from one form to another the hardest part to the interface specification task is to obtain the specification in the first part. This involves bridging the "*formality gap*" between the informal user requirements and the notation used (see below). Thus the formal notation should be as close to the user and as easy as possible for the designer (error at this stage are difficult to correct later) the user orientedness is likely to clash with implementation aims. Thus user expressive notations are not likely to be easily implementable.

[1]Windows, Icons, Mice, Pointer.

Specification vs. execution. Not only for interfaces, but for systems in general, notations and particular uses of notations that are good for specification are not necessarily good for executability. It may be easy to specify that a certain input output history is acceptable or not, but very difficult (and liable to error) to specify how an output should be derived from particular inputs. Specifications that are not executable when viewed as programs may however be executable when used as correctness checker for programs. So, for instance, the problems with backtracking in Prolog would not be a problem if the Prolog is simply run off-line to check the correctness of traces generated by some other prototyping method.

Separation vs. semantics. We have already seen that the desire of separating interface design issues from the application functionality preclude important interactive effects such as semantic feedback. In addition, they loose the essential directness of interaction.

Sequence vs. parallelism. Time and again we see that notations that are good at expressing one fail miserably at the other. This is perhaps not fundamental as much as it reflects that different schools have had different preoccupations, and different intended interface styles.

4.4 Familiarity and designer notations

Designing a system involves a translation from someone's (the client's) informal requirements to an implemented system. Once we are within the formal domain we can *in principle* verify the correctness of the system. For example, the compiler can be proved a correct transformer of source code to object and we can prove the correctness of the program with respect to the specification. Of course, what can not be proved correct is the relation between the informal requirements and the requirements as captured in the specification. This gulf between the informal requirements and their first formal statement is the formality gap [22].

For a DP application like a payroll, this may not be too much of a problem. The requirements are already in a semi-formal form (e.g., pay scales, tax laws) and they are inherently formalisable. The capture of HCI requirements is far more complex. Not only are they less formally understood to start with, but it is likely that they are fundamentally unformalisable. We are thus aiming to only formalise some aspect of a particular requirement and it is difficult to know if we have what we really want.

If this initial formalisation is wrong, then whatever follows may be worthless, it is thus essential that we do this stage as well as possible. A corollary of this is that the most important thing about a notation is how easy it is to understand and how familiar the designer is with it. However precise the notation, it is the designer's understanding of it that will be the assurance of meeting the informal requirements. Another corollary is that the first aim of such a notation is that it should match the structures of the requirements rather than being aimed at efficient implementation. A computationally poor specification can be transformed into a more efficient one, an incorrect one remains incorrect.

To summarise, what the designer is used to and feels natural with may well be the best option. In order to achieve this, it may well be best to adapt several notations to produce a notation that is tailored to the particular needs of the project. (That is a "designer" notation rather like designer fashions!)

Chapter 5

Summary evaluation

This chapter gives a summary evaluation of the various notations discussed in Chapter 1 with respect to the six criteria set out in the requirements for this report

- expressiveness
- readability
- evaluation
- manipulation
- execution
- knowledge

For quick reference a rating between 0 and 9 is given for each criteria, 0 being bad and 9 being good. However, these ratings should be treated with care as will be discussed below and to aid this a brief comment is attached to many ratings; they represent a subjective and personal assessment. The notations described within each section of Chapter 1 have largely the same ratings, thus they serve primarily to select a class suited for a particular purpose. A single table of ratings is therefore given for each class of notations rather than for each individual notation with exceptions noted in the comments.

Personal preferences within the classes are also included, although these are often based as much on familiarity and aesthetic criteria as fundamental merit.

Before beginning the summaries, we should consider some of the complexities of the criteria considered.

Expressiveness When considering expressiveness, we must consider first *what* is being described. For the notations discussed, the domain of applicability can be described as user/system or dialogue following Section 4.1.1. Further, the notations may describe these in a concrete or an abstract manner. Of these, the domain of the notation is given in the comments, whereas the rating indicates how well it achieves *within* the domain.

Readability This is inevitably a very subjective measure, depending very much on background as well as expertise.

Evaluation Obviously, this is closely related to the domain of expressiveness of the notation. One would only expect user oriented methods to give detailed cognitive measures of performance, complexity etc. Whereas one would expect dialogue centred approaches to at most give some syntactic complexity.

Manipulation I have taken this to mean the ability to analyse properties of the interface or ask questions about it given the description in the notation. I have distinguished where necessary between hand manipulation by an expert in the notation and automatic manipulation by a machine. As noted in Section 4.3 this tends to be negatively correlated with range of expressiveness. In particular, notations that are automatically analysable tend to have a small (but not necessarily unimportant) range of application.

Execution Again the meaning of this differs depending on the domain. For system oriented and dialogue descriptions, this largely corresponds to the ability to execute the specification as an early prototype, thus allowing empirical evaluation. It is possible to have such a notation that can be automatically checked against a prototype developed by alternative means, this is a form of execution and can form a bridge between the abstract but non-executable notations and the concrete fully-executable ones. However, few notations fit into this category. Finally, user oriented notations we would not expect to execute as a prototype, but may be executed as a model of the user, this being closely linked to the evaluation of cognitive demands of particular tasks.

Knowledge Obviously, it is the user-oriented methods where we would expect to see the most explicit knowledge representation of the user and task. In the GOMS school of models this is of a very limited form (simply the goal hierarchy itself) although CLG includes task and entity analysis in its top level. The user models and cognitive architectures have more detailed knowledge of the user.

When looking at dialogue description techniques and prototyping languages one must be very wary however. Several are based on similar architectures to many expert systems, for instance production systems or frames. Also, they may be implemented in languages associated with AI such as LISP or Prolog. This may be an advantage in integrating these systems with user knowledge bases but does *not* mean that they have any knowledge in themselves. I have seen one system which had so called “dialogue experts” which turned out to be simply fixed dialogue descriptions.

5.1 Psychological and soft computer science notations

Goal oriented methods

These may be used early in requirements analysis if one chooses sufficiently abstract goals with more detail being included at later stages. Typically however, they are used in the presence of an extant system.

Criteria	**Score**	**Comments**
expressiveness	7	of user
readability	3	*GOMS* a bit better
evaluation	5	short term memory capacity, complexity
manipulation	?	not very applicable
execution	5	often as user programs
knowledge	7	usually limited to goal structure

I find the more complex notations such as *CCT* virtually unreadable. There is also considerable doubt as to whether the extra complexity gives much advantage.

Grammars

These assume a detailed design of the dialogue and are thus for later stages in the design life cycle.

Criteria	Score	Comments
expressiveness	4	low level dialogue only
readability	7	
evaluation	5	some complexity
manipulation	7	easy to manipulate automatically
execution	7	limited scope
knowledge	1	

Again, the complex grammars don't seem to give you that much.

5.2 User models

If used during system design, these would have their main input when the detailed dialogue and semantics were specified (although not implemented). However, they are currently more relevant to HCI research than design.

Criteria	Score	Comments
expressiveness	8	if you believe the cognitive architectures!
readability	2	
evaluation	6	when executable, get memory use, perhaps performance
manipulation	3	
execution	6	often complex support, not all have implementations
knowledge	2–8	some (e.g., *SOAR*) extensive, others (e.g., *KLM*) little

5.3 Graphical or diagrammatic approaches

These tend to be graphical forms of linguistic or state based dialogue design notations. They are thus most suited to detailed dialogue design, well on in the life cycle. The top-down *JSD* approach could be used much earlier however.

Criteria	Score	Comments
expressiveness	6	typically little semantics
readability	9	
evaluation	5	similar to grammars or state transition forms
manipulation	7	when machine readable
execution	7	ditto
knowledge	0	

5.4 Abstract mathematical models

The models can be applied at different levels of abstraction, and thus can be used both early in the life-cycle especially in the settling of requirements, and also later on, checking specific dialogues. These are not specification notations, but may be used in conjunction with other more concrete techniques.

Criteria	Score	Comments
expressiveness	8	largely of system, not concrete however
readability	6	for the knowledgeable
evaluation	7	form basis of evaluation for other notations
manipulation	5	by hand: mathematical analysis possible
	5	automatic: some, especially checking against other notations
execution	0–4	
knowledge	4	some leeway for designer's, e.g., templates

5.5 Using general purpose formal notations

These can be used over a large range of the life-cycle. Fairly early on the requirements can (where possible) be formalised, these can then be realised in a high-level design and finally refined through various levels towards an implementation. More realistically however the latter stages tend to get far too verbose and there is mixture of functional and non-functional requirements which makes their statement difficult outside the context of a putative system design.

Criteria	Score	Comments
expressiveness	9	you *can* say almost anything, not usually of user
readability	6	for expert
evaluation	2	up to expert
manipulation	8	by hand: as much as you are able
	1	automatic: full blown theorem proving!
execution	0–9	most (e.g., *Z*) not at all, a few (e.g., *OBJ*, *me-too*) fully executable
knowledge	0	totally up to the specifier

The obvious candidate among these for researchers at or associated with the Programming Research Group at Oxford is Z. It is probably the richest specification language, including most of the constructs of standard set theory. This breadth has its problems however; I have found that user's of Z can appear to achieve a high level of competence but may lack understanding of the precise meaning of the constructs used. Also, it does not have nearly so well developed semantics compared to the equational techniques. On the other hand, their austerity can verge on the unusable.

Z also has a very flexible set of specification building constructs in the form of its schema calculus which add elegance to the resulting descriptions. Again this has its problems as the flexibility is at the price of loose semantics at the top level of the specification. For example, normally it is informally assumed than the specification consists of an initial state followed by a number of operations. This is in contrast to the equational techniques which have well behaved modular construction operators allowing proper encapsulation and parameterisation, but not the ease of use. Probably this is not too much a problem for interface specifications being more a barrier to the production of very large systems.

The problem with using a general purpose notation for dialogue specification is that the dialogue structure is not easily extractable for analysis, especially by automatic tools. (This is rather like the problem of programming dialogues in standard programming languages) A notation like Z has important uses however:

- to give precise semantics to other, more constrained notations
- as a target for translation from such notations, or to be used in a stylised form
- as the semantic part of a notation such as SPI
- to act as a carrier for models as described in chapters 2 and 3.

The generality of the notation allows these uses to be integrated with the rest of the application design.

5.6 Formal dialogue specifications

These can be used in high-level and detailed dialogue description.

Criteria	**Score**	**Comments**
expressiveness	6–8	of the dialogue, some of system semantics also
readability	6–8	
evaluation	5	grammar part analysable, also empirical
manipulation	5	
execution	9	*EPROL* and *SPI*, Marshall's not
knowledge	0	

I find SPI the most readable of these notations (with the exception of directionality of events discussed in Section 1.6). It handles both sequential and concurrent dialogues equally and is capable of linking to both specification notations and programming languages.

5.7 Non-committed prototyping notations

As prototyping notations, these are suitable for the later stages of the design life cycle (although with an iterative design approach this may actually be reached quite quickly).

Criteria	**Score**	**Comments**
expressiveness	8	concrete, of the system
readability	3–7	depends on what you are used to
evaluation	0	but again empirical
manipulation	0	with exception of pure logic and some of the functional stuff
execution	9	
knowledge	3	not intrinsic, but present in many *LISP* and *Prolog* systems

Because these notations are not aimed at interface specification one finds that the dialogue structure is hard to determine and is not easily extractable by automatic tools. If one of these languages is being used for the prototyping of the underlying application, it may be sensible to use it to prototype the dialogue in the same language. In this case, however, it would be best to use the language as a target for translation from a more dedicated notation.

5.8 Dedicated interface prototyping and development tools

These are aimed at efficient prototyping or actual implementation of the delivered system; that is in the final stages of the life-cycle.

Criteria	**Score**	**Comments**
expressiveness	6	usually only the dialogue, but link to programming languages
readability	4–7	more expressive usually implies more complex
evaluation	3	not designed for it, but often analysable
manipulation	3–6	
execution	9	
knowledge	0	

These are all basically implementation techniques. If one wishes to be able to analyse the resulting dialogue then they must be used in conjunction with other specification methods. *PAC* seems to encourage an interface oriented structure to its systems and expresses a sense of parity between input and output. It seems most suited to highly user controlled dialogues. *Input-tools* and *DICE* are both highly input-parsing oriented, their distributed dialogue control making it hard to analyse the dialogue structure.

Of the presentation tools mentioned, *Presenter* is more general, allowing the designer to create new interactive objects, whereas *FDL* covers a more restricted domain of interface styles, but it is easier to create (prescriptive) company interface styles.

5.9 UIMS and Window managers

5.9.1 UIMS

Again these belong quite late in the life cycle, perhaps more in prototyping than production systems.

Criteria	**Score**	**Comments**
expressiveness	6	
readability	3	
evaluation	0	
manipulation	0	
execution	9	
knowledge	3	sometimes share architecture with expert systems (e.g., production rules)

5.9.2 Window managers

As with UIMS, these are useful late in the life cycle.

Criteria	Score	Comments
expressiveness	6	only presentation, no dialogue
readability	3	
evaluation	0	
manipulation	0	
execution	9	
knowledge	0	

Any workstation based product would live under some window manager; however it seems far more preferable to use a cleaner tool such as *Presenter* or *FDL*. These can then be implemented over the window manager hiding some of their uglier features and letting the application interface be cleaner and easier to understand.

Chapter 6

Recommendations

This chapter looks at specific candidates for notations in the three areas:

- Task representation
- Dialogue specification
- High level requirements and standards

6.1 Task representation

To some extent this is an easier job when reverse engineering as the system exists and there is the possibility of task analysis of actual users. This means that one is even more likely to get a task/goal structure that is simply a reflection of the system than of the user's conceptual structures. However, this is always a problem with this stage of design and can be minimised by standard techniques, such as examining experts in the task domain who do not use or are new to the system.

In addition, one can express the existing dialogue as a grammar, and use this as a putative goal structure, the non-terminals corresponding to user's goals. This is clearly the goal structure implied by the system and this can be analysed by a human factors expert.

The notations described in Section 1.1 are most relevant to this stage of analysis. Of these, I find *GOMS* the most readable. However, it is of fairly limited scope, so it depends on how much effort you think is likely to be expended in this area.

This covers the goal structure, but one also may want to examine the user's model of the underlying information structures. *CLG* in its task level addresses this as does Walsh *et al.*'s *JSD* based notation [139]. Both take an entity analysis approach, not unlike that of data-base design. Task analysis approaches typically include such elements too.

For reverse engineering, one can work at several different levels: apply task analysis techniques to the user's to get their idea of the information system; work back from the interface, analysing the information as presented; and examine the information actually stored in the system.

One of the results from the work discussed in Section 2.2.4 which itself draws on some data-base analysis, was that examining what is *seen* by the user is not sufficient. It may be possible to produce a view of the underlying system and present it in such a way that the user's and designers perceptions agree. This is sufficient for viewing the data, but when one comes to update it is necessary to also agree on what is left constant by those changes. This is essentially a framing problem. As the information that is assumed constant (the *complementary view*) is not part of the view, it is easier for the user and designer to have different ideas about it without this becoming apparent (until some major error occurs because of it). Simple information system, such as flat dictionaries or tree hierarchies have obvious complementary views, but if the view becomes complex then the information analysis should include both what is seen and what is *not* seen.

6.2 Dialogue specification

Before considering specific recommendation for dialogue specification techniques, it should be emphasised that the dialogue alone is not the interaction. The usability of a system depends on the whole, immediate interface and application. Thus a dialogue specification notation that does not include application semantics may be dangerously misleading.

Having said this, it is of course acceptable to use for example, a simple grammar to obtain a high level view of the system or to examine micro-dialogue issues. This may be especially useful for certain aspects of automatic analysis. It is just that when using such a notation one should remember that it is only a small part of the story.

There are two levels of dialogue specification that should be considered. Firstly, the high-level specification of *what* the dialogue does, and then the actual implementation. The latter depends very much on the particular hardware/software system that is being used for the end system. The notations described in Section 1.8 may all be candidates as may direct coding in a general purpose language. Although in the latter case one would recommend automatic or semi-automatic translation from a higher level notation. In addition, one must consider whether a specific presentation tool is to be used, such as *Presenter* or *FDL*, or whether this is simply left to the window manager of the machine.

For higher level specification my choice would be *SPI*, mainly because I find it easiest to read (see Section 4.4 above) and because it achieves about the best semantic/syntactic separation. In addition, the flexibility about it's "host" language would enable (at various stages of design) the *eventISL* component to be written in *Z*, for example, as well as being implementable in *C* or another programming language.

The extraction of a dialogue specification from existing programs is likely to run into problems because of the syntactic/semantic description dichotomy discussed in Section 4.2.2. It will be quite easy to go through the program, noting all the I/O primatives, and performing a flow analysis. This will give a form of "grammar" for the interaction, however, it will not be clear which of the branches in the dialogue are true application ones (e.g., `if(exists(file))...`) and which are user dialogue ones (e.g., `if(in(button.pos,save_region))...`). Ideally, the majority of this work could be done by an automatic tool with a second pass by hand.

Programs running under window managers with external control (see Figure 4.1) will be considerably more difficult to process automatically. Not only are sub-dialogues likely to be dispersed throughout the event handling loop, but also the whole dialogue is 'inverted'. This sort of dialogue will need extensive hand analysis to determine what are possible dialogue traces. It is thus doubly important that the notation used for the re-engineering does not suffer from the same defect.

Even more hard to analyse are likely to be programs developed under *transaction processing* systems. The programs are required to be largely stateless, as they must cope with interleaved transactions from many terminals. Under ICL's *TP* executive and I assume under other similar systems such as IBM's *CICS*, one is able to store a certain amount of data associated with each terminal. Thus it is possible to store the current dialogue "program counter" as mode flag in this area in a similar way to the event loop in Figure 4.1. However, the temptation is not to do this, and instead to “parse” the incoming message, essentially using fields on the screen as the mode flag. The resulting programs then assume a tree structure, branching on the contents of various input fields and bears no relation to the dialogue structure. Under the ICL system, higher level, user controlled selection between the various sub-systems was coded into a control file that covered all running TP applications which would be much more easily analysable.

6.3 High level requirements and standards

The only formal approaches to this of which I am aware spring from the abstract modelling work, begun at York. These have been found capable of expressing a large range of interface issues. They must of course be used with respect for their limitations as discussed at the end of Chapter 2 and at present require a good deal of both formal and user interface expertise.

The models can be rendered into any sufficiently rich specification notation, but the models in *Z* developed at Oxford are of course ideal for people at or in contact with the Programming Research Group.

Appendix A

Section from Roger Took's thesis

A.1 Syntactic input parsing

In syntactic input parsing the *sequencing* of input events is significant. Whereas lexical parsing constructs single input tokens, syntactic parsing imposes a structure on the whole *dialogue* between user and computer. It is this capability of expressing a dialogue abstraction that makes syntactic input parsing an important feature of UIMS. Two related classes of formalism have been used: transition networks and context-free grammars.

Transition networks express allowable sequences of input by associating input tokens with transitions between system states. In its simplest form a *transition network* is a *finite state automaton* and expresses a regular grammar, although in practice the formalism is often extended to give greater power. Input parsing based on transition networks was first used by Newman in his early Reaction Handler [89], examined by Parnas [96] and Foley and Wallace [34], taken up by Boullier *et al.* in *Metavisu* [9], and by Wasserman in *USE* [140].

The transition network notation has been extended in three main ways.

1. Large networks may be modularised by allowing labels on arcs to refer to separate networks: the labelled arc may be traversed only if there is a path through the associated subsidiary network. The label can thus be viewed as a *nonterminal symbol* in a grammar. If recursive labelling is allowed, then the network has the power of a context-free grammar [60]

2. Transitions may be made to depend not simply on the current input token and state, but also on a global data structure. Transitions may enquire and update this structure. Woods [142] calls these networks Augmented Transition Networks (*ATNs*). In general, an ATN has the power of a Turing machine (since any computable function can be applied to the data structure by a transition), and this has been exploited to enable the dialogue to encapsulate all application computation, as for example in Kamran's 'Abstract Interaction Handler' [67] A more restricted form of ATN, the *pushdown automaton*, in which the data structure is limited to a stack, can implement recursion and therefore parse context-free grammars. Olsen in *SYNGRAPH* [91] and *GRINS* [95] uses a form of these called 'interactive pushdown automata'.

3. Local, independent transition networks may be embedded in a wider environment scheduled nondeterministically by input events. Jacob is most closely associated with this extension [61, 60], but Coutaz in her *PAC* model [19], and *Images* [117] have a similar scheme.

Syntactic input parsing on the basis of a context-free grammar allows a *task* abstraction. A task can be associated with a nonterminal symbol, and this can be expanded on the basis of the grammar to give the set of possible input sequences necessary to achieve the task. The grammar is conventionally specified in a variant of *BNF* productions (as, for example, in SYNGRAPH [93], or Reisner's *ROBART* languages [102]). Here is Newman's line drawing task expressed in Reisner's version of BNF ('|' is alternation, '+' is sequential concatenation. Terminal symbols are in upper case):

$$
\begin{array}{ll}
draw\_line & ::= initiate\_line + choose\_line + complete\_line \\
initiate\_line & ::= BUTTON\_PRESS + move\_cursor \\
choose\_line & ::= BUTTON\_PRESS + move\_cursor \\
complete\_line & ::= BUTTON\_PRESS \\
move\_cursor & ::= POSITION\_CURSOR \mid POSITION\_CURSOR + move\_cursor
\end{array}
$$

This grammar thus expresses a hierarchical breakdown of the task, from the top level '*draw_line*' to the terminal lexemes like '*BUTTON_PRESS*'. Note, however, that the grammar generates only sets of sequences of *terminal symbols*. In spite of the loaded symbol names, there is no semantics here – this could just as well result in a circle being drawn as a line. In order to include semantics, the grammar must essentially be 'attributed'. In SYNGRAPH, for example, Pascal procedures would be inserted into the productions of the grammar to perform the semantic operations [93]

As experience with input parsing mechanisms has grown, a number of fundamental problems have come to light. The state transition approach is incapable of handling call/return sequences: as outlined above, a labelling mechanism or pushdown state is at least required for this. As Newman points out [89], this deficiency means that semantic functions cannot be attached to *groups* of actions. Transition networks also suffer from a quadratic growth in the number of possible transitions as the number of states increases. This is a severe problem in graphical interfaces, where significant state distinctions may depend on incremental graphical changes such as moving an icon to a new location. This is compounded by the fact that the number of screen objects may vary dynamically (see Sibert *et al.*[115]). In a typical direct manipulation interface, therefore, the overall state space may be enormous. It is generally agreed that a higher than regular grammar is required to abstract and modularise the dialogue in such interfaces.

This by no means solves all problems, however. The productions of the dialogue grammar are complicated by the need to define globally-accessible user actions such as abort, help and undo. Kasik notes the difficulty of doing this [68]. Various extensions have been proposed to handle these actions. Olsen's SYNGRAPH has the notion of distinguished 'escape' and 'reenter' states for each nonterminal, which he calls 'pervasive' states [91] For example, a task is aborted via the escape state, while help might be invoked at any time via an escape and then a reenter state. Help is, of course, in addition context-sensitive. Equivalently, but more generally, Cockton [17] proposes 'Generative Transition Networks' by which transitions can be defined over sets of states, rather than state-by-state as in the standard notation. Thus an abort or help transition can easily be defined for *all* states.

Abort, undo, and general syntactic error recovery present special problems related to the parsing algorithm used. A top-down parsing algorithm commits the dialogue to a task as soon as the first possible input symbol for that task is received. The only solution for a subsequent abort, undo, or illegal input may be to backtrack to the beginning of the parse. This may be difficult if task actions such as prompting or other output have already taken place. On the other hand, a bottom-up parsing algorithm may be easier to backtrack, but provides poor intermediate feedback, since the task may not be invoked until the whole input sequence is complete. A bottom-up algorithm may be acceptable in a textual interface, where no action might be expected until the entire command string is typed and dispatched (by pushing 'return'). A direct manipulation interface, on the other hand, requires a top-down algorithm, both because users expect continuous feedback of their actions, and because a graphical screen retains no unambiguous trace of user actions (such as a command line does), over which a parser could backtrack. Green [39] Bos [137] and Kamran [66] all examine this problem. It is also interesting to note that as early as the Seillac II conference, Alan Kay was able to report [42] that experience with the well-used (textual) learning system *PLATO* had shown that error handling and back-tracking took up most of the interaction, and that finite-state grammars were unable to cope with this dialogue.

The ability to interrupt a task, get help or other information, and then return to the task at the point where it was interrupted is simply a particular case of the general need to run multiple tasks concurrently. This need is especially high in systems with interactive graphical, and particularly window-managed, interfaces. The problem from the point of view of a monolithic [73] input-parsing system like a standard UIMS is that input destined for the various tasks arrives arbitrarily interleaved: the user may type a few characters in one window, move an icon against the background, then draw a line in another window. To handle this in a *single* parse a grammar must be evolved whose set of transitions (or nonterminals) is the Cartesian product of the transitions of all the tasks. Some systems handle this interleaving complexity by disallowing it. For example, SYNGRAPH, like GKS REQUEST input, is highly moded: physical and virtual input devices are dynamically 'acquired', 'enabled', and prioritised so that inputs are delivered only in the expected contexts. A single thread of control is therefore forced on the user. The need to cater for truly *multi-threaded* dialogues, and the inadequacy of formal grammars for this, was recognised early by Alan Shaw [112] and Anson [7] (their comments even predate the flourishing of the UIMS model). Mary Shaw [113] uses the phrase 'data-driven' to convey similarly the notion of the user's freedom to update any visible data, as opposed to a 'control-driven' structure where the order of updates is determined by the program. More recently, there has been a revival of interest in the handling of multi-threaded dialogues. The fundamental perception is of the user as a real-time system – asynchronous and unpredictable [128] – and that therefore interaction should be treated as a problem in parallel computation [76].

Syntactic input parsing suffers two further fundamental objections. Firstly, whereas some use of formal grammars in input parsing is for descriptive and analytical purposes [102, 97, 85], current use in UIMS is *prescriptive*. That is, the grammar *determines* the acceptable input sequences. We can distinguish between this problem and the problem of multi-threading: in the latter a grammar restricts the number of possible concurrent dialogues, in the former a grammar restricts the number of alternative sequences in the same dialogue. Kamran [66], for example, admits that the Interaction Language of his *AIH* permits only a rigid sequencing of actions, and that more flexibility is required. It is also true that the higher the grammar, the more restrictive, in the sense that a higher grammar generates more modes.

Took [133] argues that there are two cases where dialogue determination is necessary or useful:

- when there is a necessary sequencing in the operations provided by the functionality, for example non-commutative operations like pushing and then popping an empty stack, or logging in and then opening a file.

- when one of the participants in the dialogue cannot be expected to be responsible for its actions, for example a novice user who does not know that exiting from an editor does not automatically save his edited file, or a nuclear reactor that does not 'know' that raising its damping rods and voiding its coolant would result in a melt-down. In these cases it would be useful to impose *external* temporal or logical constraints on the possible traces of the functionality for the good of the participants. It is not clear that a grammar is the best formalism for doing this.

Apart from these, there are also cases where dialogue determination is unnecessary, for example in the ordering of parameters to an operation. A just balancing of these factors should result in what Thimbleby [130] calls a *well-determined* dialogue.

It is nevertheless a characteristic of successful direct manipulation interfaces that they are user-driven (data driven) rather than control-driven. Shneiderman's justification [114] for allowing the user this freedom is that any operation in such an interface should be easily reversible. Grammar-based systems have responded to the need for user freedom by introducing nondeterminism in their productions (see, for example [137] and [108]). This, however, is a tacit admission that parsing input according to a grammar may not be appropriate for direct manipulation.

Finally, as Reisner points out [102], not all syntactically correct dialogues allowed by a grammar-driven input parser may be legal in terms of the underlying task. That is, there may be *semantic* errors not trapped by the input parser. The example above of popping an empty stack is a case in point. There are two approaches that may be taken in this case. In one, errors may simply be allowed to pass through to the application task, which may then need to instigate a special dialogue with the user in order to correct them. This strategy is adopted by GWUIMS [116]. An alternative approach is to allow the dialogue knowledge of or communication with the application task. For example, parameter types may be declared in advance against which the dialogue can check input, or enquiry operations may be allowed on the task state. However, it may be difficult to abstract the dialogue from the task without duplicating much (in the limit, all) of its functionality. The general problem is that of 'semantic feedback', either to report errors, or to prevent them.

Parsing user input according to a grammar, therefore, has many theoretical drawbacks. In practice, also, experience of using grammars has not been positive. Two complaints are voiced. Firstly, specifying the dialogue in a separate language or formalism from the application functionality is often difficult. SYNGRAPH was not widely used for this reason [94]. The only real solution is to generate the dialogue automatically. Green [40] proposes this, but there are few prototypes [87]. Secondly, parsing user input according to the grammar often presents problems. Hekmatpour and Woodman complain of this [52]. In recent papers, Olsen, Hudson, and Hill have strongly criticised the syntactic approach to input. Olsen [94] thinks that ease of use is often more critical to the success of a UIMS than syntactic capability. Having used syntactic input parsing in the SYNGRAPH and GRINS, Olsen's latest system, *MIKE* [92], abandons the syntactic component. Coutaz similarly abandons the single dialogue component in her PAC model [19]. Hudson [58] views syntactic input as reducing 'engagement' in a direct manipulation system, since the user is communicating with the system rather than with the objects of interest, and concludes that syntax should be minimised. Hill [54] regards the parser-based approach as "clumsy and awkward", and argues for a user interface specification language with programming power. However, there is a danger in this last viewpoint of starting on another Wheel of Reincarnation which goes: programming languages – input devices – virtual input devices – grammars – programming languages – input devices ...

We conclude that syntax parsing according a separable grammatical specification is not a viable interface service for direct manipulation interfaces. This does not mean to say that anarchy will therefore prevail in dialogues. Whenever input sequencing is necessary, this can best be determined and monitored by the underlying functionality. What is minimally required in a multi-tasking environment is an unambiguous mechanism for input dispatching which is not deeply moded. Graphical, rather than syntactic, structures provide an ideal 'switchboard' for allowing the user to direct input selectively and without unnecessary sequencing constraints. In terms of input, the model in this thesis therefore concentrates on facilities at the presentation, rather than at the dialogue, level. It uses events, and constructs abstract events such as region collision, but is not prescriptive as to how presentation is linked to application semantics.

A.1.1 Events

The usual input formalism for handling multi-threading is the event, along with an associated event handler. The basic structure is thus a set of $\langle input, action \rangle$ pairs [113]. Green [39] shows that this is greater in expressive power than either state-transition networks or context-free grammars. This, however, is purely because the event *handlers* are allowed programming constructs. There is thus little or no notion of a syntax over the events themselves other than what might be imposed by individual event handlers – events are simply directed as they arrive to matching handlers. This is what gives the model its flexibility and frees it from the restrictions of syntactic input parsing.

There are some hybrids, though, in which tasks are allowed to maintain their own input syntax independently of other tasks. As noted above, Jacob [61] expresses task syntax using state-transition networks, but his top-level input is event-driven. When an individual task is suspended (because the current input does not match any of its possible transitions) it maintains its state until control returns. Equivalently, Scott and Yap [108] express task syntax as a context-free grammar, but allow parallel invocations of tasks. The models of van den Bos [135], and ten Hagen [129] allow the expression of an 'input rule' against which input must be matched before an event is triggered. All these event-driven models thus depend critically on top-level input dispatching, but often the precise mechanics of this are left vague. In Green's model, handlers express interest in certain types of input token. In Jacob's, events are directed according to which task is able to accept the input. In Scott's, an explicit 'context' is associated with each event, and the individual grammars refer to events only within a particular context. It is obvious that many event-driven systems assume that primary input dispatching is spatial, based on windows or icons, and that there may be an 'interest' mechanism, as in X or NeWS ('sensors' in InterViews [75]). Green's more general event-driven model also allows event handlers to generate events, which brings it close to the communication model of NeWS processes and the object-oriented paradigm. In this form, the event-driven model can be expressed as a *production system* [57], although this term must be distinguished from the 'productions' of a formal grammar. Hopgood and Duce, for example, give a production system for Newman's line drawing task:

$B1$ −> ⟨enable tracking device⟩
X −> ⟨display cursor⟩
$B2$ −> ⟨store start point⟩ S
$S\,X$ −> ⟨display rubber band line⟩ S
$B3$ −> ⟨store end point⟩

In this system, productions (held in Long Term Memory (LTM)) are invoked on each time interval if the events to the left of the arrow are present in Short Term Memory (STM). The events are not ordered. X is the position of the cursor, which is generated on each time interval. $B1$, $B2$, and $B3$ are button events. The S event after the action specification is generated by the rule and written back to STM. Events are consumed on each time interval, but may match more than one rule. This formalism is thus more expressive than either the transition network or BNF grammar given above. For example, if S and X are in STM then both the second and the fourth rules are satisfied. The formalism is also less moded. For example, the end point could be given before the start point (by pressing $B3$ before $B2$). A refinement of the production system imposes an ordering on the production rules in LTM, which reduces the ambiguity but increases the modedness.

Green in the University of Alberta UIMS [38], Cardelli and Pike [13], Tanner [127], Hill [54], and Lantz [73] have all produced systems or formalisms for handling user input based on events. A useful benefit of the event-driven model, exploited by Hill, is that it can also handle concurrent *multi-device* input. That is, so long as processes are allowed to generate events, any device can be read by a monitoring process which can then generate appropriate events on its behalf.

Acknowledgements

This report was originally issued as a technical note[1] on the ESPRIT II "REDO" project. The overall project manager, Dr. Panayotis Katsoulakos of Lloyd's Register of Shipping, gave permission for it to be used outside the project, and in particular for it to be issued in this form.

Michael Harrison at York University agreed to help produce this report. As a result, Alan Dix *et al.* at York University wrote the majority of the contents. Appendix A is an extract from Roger Took's Ph.D. thesis. The cooperation of the HCI (Human Computer Interaction) group at York University is greatly appreciated; without it this report would not have been possible.

Dr. Henk van Zuylen of Delft Hydraulics provided some of the documents which were used as input to the report. He also supplied comments which were used in finalising the report.

Gregory Abowd of the Programming Research Group at Oxford wrote Chapter 3 and also provided comments on the report. Jonathan Bowen coordinated the authors, added the index, and compiled and edited the final document.

The bibliography overleaf is a combination of references collected by Gregory Abowd and the HCI group at York. Jonathan Bowen merged the two (and added a few extra ones as well). Please note that important references are marked with "$***$" and are particularly recommended for further reading.

This report has been prepared using the LaTeX document preparation system. The Z specifications have been formatted and type-checked using the *f*UZZ package.

Michael Harrison and Roger Took are lecturers in the Department of Computer Science at the University of York. Gregory Abowd is a D.Phil. student at the Programming Research Group. Alan Dix is an SERC Post-doctoral Fellow at the University of York. Jonathan Bowen is a Research Officer on the SERC-funded Software Engineering Project at the Programming Research Group. The support of the SERC is gratefully acknowledged.

[1] Document number 2487-TN-PRG-1008, August 1989

Bibliography

[1] H. Alexander. ECS – a technique for the formal specification and rapid prototyping of human-computer interaction. In M.D. Harrison and A. Monk, editors, *People and Computers: Designing for usability*, pp. 157–179, Cambridge University Press, 1986.

[2] H. Alexander. Executable specifications as an aid to dialogue design. In Bullinger and Shackel, editors, *Proceedings of INTERACT '87*, North Holland, 1987.

[3] H. Alexander. Formally-based techniques for designing human-computer dialogues. In Diaper and Winder, editors, *People and Computers III*, pp. 201–214, Cambridge University Press, 1987.

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

[5] J.L. Alty. The application of path algebras to interactive dialogue design. *Behaviour and Information Technology*, **3**(2):119–132, 1984.

[6] S.O. Anderson. Proving properties of interactive systems. In Harrison and Monk, editors, *People and Computers: Designing for usability*, Cambridge University Press, 1986.

[7] E. Anson. The semantics of graphical input. *ACM Computer Graphics*, **13**(2):113–120, January 1979.

[8] P. Barnard and M.D. Harrison. Integrating cognitive and system models in human computer interaction. In Sutcliffe and Macauley, editors, *People and Computers V*. Cambridge University Press, 1989.

[9] P. Boullier, J. Gros, P. Jancene, A. Lemaire, F. Prusker, and E. Saltel. Metavisu: a general purpose graphics system. In F. Nake and A. Rosenfield, editors, *Graphic Languages*, pp. 244–267, North-Holland, 1972.

[10] J.P. Bowen. Formal specification of window systems. PRG-74 Technical Monograph, Programming Research Group, Oxford University Computing Laboratory, June 1989.

[11] S.K. Card, T.P. Moran, and A. Newell. The keystroke-level model for user performance with interactive systems. *Comm. ACM*, **23**:396–410, 1980.

[12] S.K. Card, T.P. Moran, and A. Newell. *The Psychology of Human Computer Interaction.* Lawrence Erlbaum, 1983.

[13] L. Cardelli and R. Pike. Squeak – a language for communicating with mice. *ACM Computer Graphics*, **19**(3):199–204, 1985.

[14] U.H. Chi. Formal specification of user interfaces: a comparison and evaluation of four axiomatic approaches. *IEEE Trans. on Software Engineering*, SE-**11**(8):671–685, 1985.

[15] G. Cockton. Where do we draw the line? – derivation and evaluation of user interface software separation rules. In Harrison and Monk, editors, *People and Computers: Designing for usability*, pp. 417–432, Cambridge University Press, 1986.

[16] G. Cockton. Some critical remarks on abstractions for adaptable dialogue managers. In Diaper and Winder, editors, *People and Computers III.* Cambridge University Press, 1987.

[17] G. Cockton. Generative transition networks: a new communication control abstraction. In Jones and Winder, editors, *People and Computers IV*, pp. 509–528, Cambridge University Press, 1988.

[18] S. Cook. Modelling generic user-interfaces with functional programs. In Harrison and Monk, editors, *People and Computers: Designing for usability*, Cambridge University Press, 1986.

[19] J. Coutaz. Pac, an object oriented model for dialog design. In H.J. Bullinger and B. Shackel, editors, *Human-Computer Interaction – INTERACT '87 (Participants' Edition)*, pp. 431–436, North-Holland, 1987.

[20] J. Coutaz. Architecture models for interactive software. Technical report, Laboratoire de Genie Informatique, Grenoble Cedex, France, January 1989.

[21] B. Curtis. The crucible of a new discipline. In [109], pp. 67–72,

[22] A.J. Dix. Abstract, generic models of interactive systems. In D.M. Jones and R. Winder, editors, *People and Computers IV: Proc. HCI '88*, pp. 63–77, Cambridge, January 1988.

[23] A.J. Dix. Formal methods and interactive systems: principles and practice. Technical report, University of York, 1987. Ph.D. Thesis.

[24] A.J. Dix. The myth of the infinitely fast machine. In Diaper and Winder, editors, *People and Computers III: Proc. HCI '87*, Cambridge University Press, 1987.

[25] A.J. Dix and M.D. Harrison. Principles and interaction models for window managers. In M.D. Harrison and A.F. Monk, editors, *People and Computers: Designing for usability*, pp. 352–366, Cambridge University Press, 1986.

[26] A.J. Dix and M.D. Harrison. Interactive systems design and formal development are incompatible? In McDermid, editor, *Proceedings 1988 Refinement Workshop.* Butterworth Scientific, 1989.

[27] A.J. Dix, M.D. Harrison, C. Runciman, and H.W. Thimbleby. Interaction models and the principled design of interactive systems. In Nichols and Simpson, editors, *European Software Engineering Conference*, Springer Lecture Notes, 1987.

[28] * * * A.J. Dix and C. Runciman. Abstract models of interactive systems. In Johnson and Cook, editors, *People and Computers: Designing the interface*, Cambridge University Press, 1985.

[29] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer, 1985.

[30] D. England. A user interface design tool. In *ESEC'87 – 1st European Software Engineering Conference*, Springer-Verlag, 1987.

[31] D. England. Graphical prototyping of graphical tools. In Jones and Winder, editors, *People and Computers IV*, pp. 407–420, Cambridge University Press, 1988.

[32] M.U. Farooq and W.D. Dominick. A survey of formal tools and models for developing user interfaces. *International Journal of Man-Machine Studies*, **29**, 1988, pp. 479–496,

[33] G. Fischer. Human-computer interaction software: lessons learned, challenges ahead. *IEEE Software*, January 1989, pp. 44–52,

[34] J.D. Foley and V.L. Wallace. The art of natural graphic man-machine conversation. *Proc. IEEE*, **62**(4):462–471, April 1974.

[35] J.D. Foley, W.C. Kim, and C.A. Gibbs. Algorithms to transform the formal specification of a user-computer interface. In Bullinger and Shackel, editors, *Proceedings of INTERACT '87*, pp. 1001–1006, North Holland, 1987.

[36] J. Foley, Won Chul Kim, S. Kova Kimčević, and K. Murray. Defining Interfaces at a High Level of Abstraction. *IEEE Software*, January 1989, pp. 25–32,

[37] A. Goldberg. *Smalltalk-80, The interactive programming environment.* Addison-Wesley, 1984.

[38] M. Green. Report on dialogue specification tools. In G.E. Pfaff, editor, *User Interface Management Systems*, pp. 9–20, Springer-Verlag, Berlin, January 1985.

[39] M. Green. A survey of three dialogue models. *ACM Trans. on Graphics*, **5**(3):244–275, January 1986.

[40] M. Green. Directions for user interface management systems research. *ACM Computer Graphics*, **21**(2):113–116, April 1987.

[41] T.R.G. Green, F. Schiele, and S.J. Payne. Formalisable models of user knowledge in human-computer interaction. In *Working with Computers*, Academic Press, 1988.

[42] R.A. Guedj, P.J.W. ten Hagen, F.R.A. Hopgood, H.A. Tucker, and D.A. Duce, editors. *Methodology of Interaction*, North-Holland, 1980.

[43] F.G. Halasz, T.P. Moran, and R.H. Trigg. Notecards in a nutshell. In Carroll and Tanner, editors, *Human Factors in Computer Systems & User Interface (CHI '87)*, 1987.

[44] W.J. Hansen. User engineering principles for interactive systems. In *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, McGraw-Hill, New York, 1984, pp. 217–231,

[45] M.D. Harrison and A.J. Dix. Modelling the relationship between state and display in interactive systems. In Tauber and Gorny, editors, *Visualisation in Human Computer Interaction*, Springer Lecture Notes, 1988.

[46] M.D. Harrison and A.J. Dix. A state model of direct manipulation. In Harrison and Thimbleby, editors, *Formal Methods in Human Computer Interaction*, pp. 135–156, Cambridge University Press, 1989.

[47] M.D. Harrison, C.R. Roast, and P.C. Wright. Complementary methods for the iterative design of interactive systems. In *Proceedings HCI International '89, Boston*. Elsevier Scientific, 1989.

[48] M.D. Harrison and H. Thimbleby (editors). *Formal Methods in HCI*. Cambridge University Press, to be published.

[49] M.D. Harrison, and H. Thimbleby. Formalising Guidelines for the Design of Interactive Systems. In Johnson and Cook, editors, *People and Computers: Designing the interface*, Cambridge University Press, 1985.

[50] R. Harston. User-Interface Management Control and Communication. *IEEE Software*, January 1989, pp. 62–70,

[51] S. Hekmatpour and D. Ince. Evolutionary prototyping and the human-computer interface. In Bullinger and Shackel, editors, *Proceedings of INTERACT '87*, pp. 479–484, North Holland, 1987.

[52] S. Hekmatpour and M. Woodman. Formal specification of graphical notations and graphical software tools. Technical report, Open University, 1987.

[53] A. Henderson. The trillium user interface. In *Proceedings of CHI'86 – Human factors in computing systems*, 1986.

[54] R.D. Hill. Event-response systems – a technique for specifying multi-threaded dialogues. In *Proc. SIGCHI+GI '87: Human Factors in Computing Systems*, pp. 241–248, Toronto, Canada, January 1987.

[55] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, London, 1985.

[56] S.J. Holmes. Overview and user manual for doubleview. Technical Report YCS109, Department of Computer Science, University of York, 1989.

[57] F.R.A. Hopgood and D.A. Duce. A production system approach to interactive graphic program design. In R.A. Geudj, P.J.W. ten Hagen, F.R.A. Hopgood, H.A. Tucker, and D.A. Duce., editors, *Methodology of Interaction*, pp. 247–263, North-Holland, 1980.

[58] S.E. Hudson. UIMS support for direct manipulation interfaces. *ACM Computer Graphics*, **21**(2):120–124, January 1987.

[59] R.J.K. Jacob. Using formal specifications in the design of a human-computer interface. *Communications of the ACM*, **26**:4, pp. 259–264, April 1983.

[60] R.J.K. Jacob. A visual programming environment for designing user interfaces. In Shi-Kuo Chang, Tadao Ichikawa, and P.A. Ligomenides, editors, *Visual Languages*, pp. 87–107, Plenum Press, New York, 1986.

[61] R.J.K. Jacob. A specification language for direct manipulation user interfaces. *ACM Transactions on Graphics*, **5**(4), 1986.

[62] P. Johnson, D. Diaper, and J.B. Long. Task analysis in interactive systems design and evaluation. In G. Johannsen, C. Mancini, and L. Martensson, editors, *Analysis, design and evaluation of man-machine systems*, Pergamon Press, 1985.

[63] C. Johnson. Temporal logic applied to interaction. Technical report, Human Computer Interaction Group, York University, April 1989.

[64] S.C. Johnson. Yacc: Yet another compiler compiler. Computing Science Technical Report 32, Murray Hill, 1975.

[65] C.B. Jones. *Software development: a rigorous approach.* Prentice-Hall International, London, 1980.

[66] A. Kamran. Issues pertaining to the design of a user interface management system. In G.E. Pfaff, editor, *User Interface Management Systems*, pp. 43–48, Springer-Verlag, 1985.

[67] A. Kamran and M.B. Feldman. Graphics programming independent of interaction techniques and styles. *ACM Computer Graphics*, **17**(1):58–66, January 1983.

[68] D.J. Kasik, M.A. Lund, and H.W. Ramsey. Reflections on using a UIMS for complex applications. *IEEE Software*, **6**(1):54–61, January 1989.

[69] D.E. Kieras and P.G. Polson. An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies*, **22**:365–394, 1985.

[70] G. Kiss and R. Pinder. The use of complexity theory in evaluating interfaces. In Harrison and Monk, editors, *People and Computers: Designing for usability*, pp. 447–463, Cambridge University Press, 1986.

[71] C. Knowles. Can cognitive complexity theory (CCT) produce an adequate measure of system usability? In Jones and Winder, editors, *People and Computers IV*, pp. 291–307, Cambridge University Press, 1988.

[72] J.E. Laird, A. Newell, and P. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence*, **33**:1–64, 1987.

[73] K.A. Lantz *et al.* Reference models, window systems, and concurrency. *ACM Computer Graphics*, **21**(2):87–97, April 1987.

[74] R. Van Liere and H.J. Schouten. The DICE language and cookbook.

[75] M.A. Linton and R.R. Calder. The design and implementation of interviews. In *Proc. USENIX C++ Workshop*, pp. 256–267, Santa Fe, New Mexico, USA, 1987.

[76] W.R. Mallgren. *Formal Specification of Interactive Graphics Programming Languages.* MIT Press, 1983.

[77] L. Marshall. Ph.D. thesis. Technical report, University of Manchester, 1986.

[78] S. McGregor and J. Parodi. *DECwindows: architectural overview.* Version 1.1, Digital Equipment Corporation, USA, 1988.

[79] A. McLean. Human factors and the design of user interface management systems. *Information and Software Technology*, **29**(4):192–201, 1987.

[80] A. McLean, P. Barnard, and M. Wilson. Rapid prototyping of dialogue for human factors research: the Easie approach. In M.D. Harrison and A. Monk, editors, *People and Computers: Designing for usability*, pp. 180–195, Cambridge University Press, 1986.

[81] S. Minor. Structured command interaction based on grammar interpreting synthesiser. In Bullinger and Shackel, editors, *Proceedings of INTERACT '87*, pp. 731–737, North Holland, 1987.

[82] A.F. Monk. Mode errors: a user-centred analysis and some preventative measures using keying contingent sound. *International Journal of Man-Machine Studies*, **24**, 1986.

[83] A.F. Monk and A.J. Dix. Refining early design decisions with a black-box model. In *People and Computers III: Proc. HCI '87.* Cambridge University Press, 1987.

[84] A.F. Monk. A procedure for identifying unpredictability, unnecessary complexity, inconsistency and effects which are hard to reverse. Paper submitted for inclusion in CHI '87.

[85] T.P. Moran. The command language grammar: a representation for the user interface of interactive systems. *International Journal of Man Machine Systems*, 15, 1981.

[86] C.C. Morgan. *Programming from specifications.* Prentice-Hall International, London, 1989.

[87] B.A. Myers. Tools for creating user interfaces: an introduction and survey. Technical Report CMU-CS-88-107, CMU, January 1988.

[88] B.A. Myers. User-interface tools: introduction and survey. *IEEE Software*, January 1989, pp. 15–23,

[89] W.M. Newman. A system for interactive graphical programming. In *AFIPS Spring Joint Computer Conference*, pp. 47–54, 1968.

[90] D.A. Norman. Cognitive engineering principles in the design of human-computer interfaces. In [109], pp. 11–16,

[91] D.R. Olsen. Pushdown automata for user interface management. *ACM Trans. Graphics*, **3**(3):177–203, July 1984.

[92] D.R. Olsen. Mike: the Menu Interaction Kontrol Environment. *ACM Trans. Graphics*, **5**(4):318–344, October 1986.

[93] D.R. Olsen and E.P. Dempsey. Syngraph: a graphic user interface generator. *ACM Computer Graphics*, **17**(3):43–50, July 1983.

[94] D.R. Olsen. Larger issues in user interface management. *ACM Computer Graphics*, **21**:134–137, 1987.

[95] D.R. Olsen Jnr., E.P. Dempsey, and R. Rogge. Input/output linkage in a user interface management system. *ACM Computer Graphics*, **19**(3):191–197, January 1985.

[96] D.L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings 24th National ACM Conference*, pp. 379–385, 1969.

[97] S.J. Payne. Task-action grammars. In B. Shackel, editor, *Human-Computer Interaction, Interact '84*, pp. 527–532, North-Holland, 1985.

[98] S.J. Payne and T.R.G. Green. Task-action grammars: a model of mental representation of task languages. *Human-Computer Interaction*, **2**(2):93–133, 1986.

[99] 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 It's Application*, pp. 175 – 187. Ablex Publishing, Norwood, New Jersey, USA, 1986.

[100] G.E. Pfaff, editor. *User Interface Management Systems.* Springer, 1985.

[101] M.J. Plasmeijer. Input tools – a language model for interaction and process communication. Technical report, de Katholieke Universiteit to Nijmegen, 1981.

[102] P. Reisner. Formal grammar and human factors design of an interactive graphics system. *IEEE Transactions on Software Engineering*, SE-**7**(2):229–240, 1981.

[103] C.R. Roast and M.D. Harrison. The specification and prototyping of interaction models using a dynamic logic. Draft, Human Computer Interaction Group, University of York, April 1989.

[104] T.L. Roberts. Perspectives of a Modern User-Interface Designer. In [109], pp. 61–66,

[105] C. Runciman, M. Firth, and N. Jagger. Preserving interactive behaviour through transformation. Technical report, Department of Computer Science, University of York, 1988.

[106] C. Runciman and N.V. Hammond. User programs: a way to match computer system design and human cognition. In Harrison and Monk, editors, *People and Computers: Designing for usability*, pp. 464–481, Cambridge University Press, 1986.

[107] Colin Runciman. From abstract interaction models to functional prototypes. In M.D. Harrison and H. Thimbleby, editors, *Formal methods in Human Computer Interaction*, Cambridge University Press, 1989.

[108] M.L. Scott and S.-K. Yap. A grammar-based approach to the automatic generation of user interface dialogues. In E. Solloway, D. Frye, and S.B. Sheppard, editors, *Proc. CHI '88*, pp. 73–78, Addison Wesley, January 1988.

[109] G. Salvendy (editor). *Human-Computer Proceedings of the First U.S.A-Japan Conference on Human-Computer Interaction, Honolulu, Hawaii, August 18–20, 1984.* Elsevier, Amsterdam, 1984.

[110] B. Sharratt. Top-down interactive systems design: some lessons learnt from using command language grammar specifications. In Bullinger and Shackel, editors, *Proceedings of INTERACT '87*, pp. 395–399, North Holland, 1987.

[111] B.D. Sharratt. The incorporation of early interface evaluation into command language grammar specifications. In Diaper and Winder, editors, *People and Computers III: Proc. HCI '87*, Cambridge University Press, 1987.

[112] A.C. Shaw. On the specification of graphics command languages and their processors. In Guedj *et al.*, editors, *Methodology of Interaction*, pp. 160–171, North-Holland, 1980.

[113] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch. *Descartes: a programming-language approach to interactive display interfaces*, volume 0-89791-108-3/83/006/0100. ACM, 1983.

[114] B. Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, **16**(8):57–69, 1983.

[115] J. Sibert, R. Belliardi, and A. Kamran. Some thoughts on the interface between user interface management systems and application software. In G.E. Pfaff, editor, *User Interface Management Systems*, pp. 183–192, Springer-Verlag, 1985.

[116] J.L. Sibert, W.D. Hurley, and T.W. Bleser. An object-oriented user interface management system. *ACM Computer Graphics*, **20**(4):259–268, August 1986.

[117] L.P. Simoes and J.A. Marques. Images – an object oriented UIMS. In H.-J. Bullinger and B. Shackel, editors, *Human-Computer Interaction – Interact '87*, pp. 751–756, North-Holland, 1987.

[118] T. Simon. Analysing the scope of cognitive models in human-computer interaction: a trade-off approach. In Jones and Winder, editors, *People and Computers IV*, pp. 79–93, Cambridge University Press, 1988.

[119] T. Simon and R.M. Young. GOMS mets STRIPS: the integration of planning with skilled procedure execution in human-computer interaction. In Jones and Winder, editors, *People and Computers IV*, pp. 581–594, Cambridge University Press, 1988.

[120] J.M. Spivey, *Understanding Z, a Specification Language and its Semantics.* Cambridge University Press, 1987.

[121] ∗∗∗ J.M. Spivey. *The Z notation: a reference manual.* Prentice Hall International, London, 1989.

[122] H.L. Stern. Comparison of Window Systems. *BYTE*, November 1987, pp. 265–272,

[123] B.A. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, **1**:157–202, 1982.

[124] B.A. Sufrin. Formal methods and the design of effective user interfaces. In M.D. Harrison and A. Monk, editors, *People and Computers: Designing for usability*, pp. 24–43, Cambridge University Press, 1986.

[125] ∗ ∗ ∗ B.A. Sufrin and Jifeng He. Specification, refinement and analysis of interactive processes. In Harrison and Thimbleby, editors, *Formal methods in Human Computer Interaction*, Cambridge University Press, 1989.

[126] A. Sutcliffe. Some experiences in integrating specification of human-computer interaction within a structured system development method. In Jones and Winder, editors, *People and Computers IV*, pp. 145–160, Cambridge University Press, 1988.

[127] P.P. Tanner. Multi-thread input. *ACM Computer Graphics*, **21**(2):142–145, January 1987.

[128] P.P. Tanner, S.A. MacKay, D.A. Stewart, and M. Wein. A multitasking switchboard approach to user interface management. *ACM Computer Graphics*, **20**(4):241–248, August 1986.

[129] P.J.W. ten Hagen and J. Derksen. Parallel input and feedback in dialogue cells. In G.E. Pfaff, editor, *User Interface Management Systems*, pp. 109–124, Springer-Verlag, 1985.

[130] H.W. Thimbleby. Dialogue determination. *International Journal of Man Machine Systems*, **13**:295–304, 1980.

[131] * * * H.W. Thimbleby. Generative User-Engineering Principles for User Interface Design. In *Human-Computer Interaction – Interact '84*, B. Shackel, editor, North-Holland, Amsterdam, 1985, pp. 661–666,

[132] R.K. Took. The presenter – a formal design for an autonomous display manager. In I. Sommerville, editor, *Software Engineering Environments*, pp. 151–169, 1986.

[133] R.K. Took. *Surface interaction: a formal model for the presentation level of applications and documents*. Ph.D. thesis, University of York, 1989, in preparation.

[134] J.J.P. Tsai and J.C. Ridge. Intelligent Support for Specifications Transformation. *IEEE Software*, November 1988, pp. 28–35,

[135] J. van den Bos. Whither device independence in interactive graphics? *International Journal of Man-Machine Studies*, **18**:89, 1983.

[136] J. van den Bos and R. Plasmeijer. Input-output tools: a language facility for interactive and real-time systems. *IEEE Transactions on Software Engineering*, SE-**9**(3):247–259, 1983.

[137] J. van den Bos. High-level graphic input tools and their semantics. In R.A. Geudj, P.J.W. ten Hagen, F.R.A. Hopgood, H.A. Tucker, and D.A. Duce., editors, *Methodology of Interaction*, pp. 159–169, North-Holland, 1980.

[138] M. van Harmelen and S.M. Wilson. VIZ: a production system based user interface management system. In *Eurographics'87*. Elsevier Science, 1987.

[139] P. Walsh, K. Lim, J. Long, and M. Carver. JSD and the design of user interface software. *Ergonomics*, 1989.

[140] A.I. Wasserman. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, SE-**11**(8):699–713, 1985.

[141] A.I. Wasserman and D.T. Shewmake. The role of prototypes in the user software engineering (use) methodology. In H.R. Hartson, editor, *Advances in Human-Computer Interaction, Volume 1*. Ablex, 1984.

[142] W.A. Woods. Transition network grammars for natural language analysis. *Communications ACM*, **13**(10), 1970.

[143] R.M. Young, P. Barnard, T. Simon, and J. Whittington. How would your favourite user model cope with these scenarios? *SIGCHI Bulletin*, **20**:51–55, 1989.

[144] R.M. Young and T.R.G. Green. Programmable user models as aids to the interface designer. 1988, in preparation.

Index