FORMALLY-BASED TOOLS AND TECHNIQUES FOR HUMAN-COMPUTER DIALOGUES

by

Heather Alexander

Submitted to the University of Stirling
in partial fulfilment of requirements for
the degree of Doctor of Philosophy

ABSTRACT

With ever cheaper and more powerful technology, the proliferation of computer systems, and higher expectations of their users, the user interface is now seen as a crucial part of any interactive system. As the designers and users of interactive software have found, though, it can be both difficult and costly to create good interactive software. It is therefore appropriate to look at ways of "engineering" the interface as well as the application, which we choose to do by using the software engineering techniques of specification and prototyping.

Formally specifying the user interface allows the designer to reason about its properties in the light of the many guidelines on the subject. Early availability of prototypes of the user interface allows the designer to experiment with alternative options and to elicit feedback from potential users.

This thesis presents tools and techniques (collectively called SPI) for specifying and prototyping the dialogues between an interactive system and its users. They are based on a formal specification and rapid prototyping method and notation called me too, and were originally designed as an extension to me too. They have also been implemented under UNIX*, thus enabling a transition from the formal specification to its implementation.

* UNIX is a trademark of AT&T Bell Laboratories

ACKNOWLEDGEMENTS

My thanks to:

Peter Henderson, without whose encouragement, ideas, criticism and enthusiasm this work would not have been started, continued or finished;

ICL in general, and Tony Gale in particular, for supporting me financially and allowing me the time to pursue this research;

Bob Clark and Simon Jones for their advice, criticism and time spent in proof-reading this thesis, and to Val, Cyd and Simon for additional proof-reading;

Lynne for typing this thesis, displaying patience and fortitude in the face of many last-minute changes and an uncooperative word-processor;

my family for all their encouragement;

and to some very special people who have opened their lives to me and kept me moving forwards - Rena, Billy, Fiona, Lynn and Catherine.

imprimis gratia deo

CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

Until recently, most of the emphasis in software development has been on the functionality of the end-product rather than its user interface. However, with cheaper and more powerful technology, such as high-resolution screens and speech processing, and the proliferation of computer systems (particularly of small, low-cost systems), the user interface is now seen as a crucial part of any interactive system.

As developers of interactive software have found, though, it can be both difficult and costly to create good interactive software, with production of the actual interactive portion of the software consuming the major part of the development effort, as reported in [Sutton & Sprague 78]. It is therefore appropriate to look at ways of "engineering" the interface as well as the application.

Formal specification and rapid prototyping are two software engineering techniques advocated as means of improving the process of software development in general; we wish to apply them to the design of software for human-computer interaction. Formally specifying the user interface allows the designer to reason about its properties in the light of the many guidelines on the subject [Dix & Runciman 85]. Early availability of prototypes of the user interface allows the designer to experiment with alternative options and to elicit feedback from potential users, eg. [Bournique & Treu 85].

This thesis presents SPI, a way of specifying and prototyping the dialogue between a system and its user. SPI encompasses both a method and languages, and is implemented by a system which allows dialogue specifications to be executed as prototypes.

In this introduction, we begin by clarifying some of the problems of software development in general before going on to examine the particular difficulties related to human-computer interaction. The techniques of formal

specification and rapid prototyping are introduced, together with one particular way of drawing the two together which we have exploited in this work.

1.1 Software development

Software development is widely acknowledged to be an expensive, time-consuming and error-prone undertaking (see, for example, [Jensen & Tonies 79] [Sommerville 82]). This has prompted calls for a more scientific or engineering-based approach [Jones 80, 86] [Hoare 82a]. "Software engineering" is the name given to the discipline which began to emerge in the late 1960's in response to the problems encountered as new, more powerful computer technologies became available. Despite considerable research and development devoted to software engineering, the term "software crisis" is still applied to this situation [Pressman 82]. There are many reasons for the problems we find in producing reliable software. Some arise from the nature of software itself while some are due to the way in which it is developed. The situation is aggravated by the growing demand for reliable, high-quality and increasingly complex software.

Unlike other engineering disciplines, the raw material in software engineering is abstract rather than concrete, logical rather than physical, concerned with ideas, algorithms and structure. Software deals with quantities that take discrete values, rendering interpolation and extrapolation invalid. This makes it harder to verify the correct operation of the software product by testing, since testing it at the limits of permissible values is insufficient to guarantee its behaviour between those limits. In addition, there are a great many such quantities in any software product, and determining the effects of all potential interactions between them becomes effectively impossible.

With the increasing availability of computer technology has come two demands from its users. Firstly, the widespread use of computers has led to a growing dependence on them in many areas of society, such as business, defence, and medical systems, where failure of a system can result in failure of a company or loss

of life. Consequently, one demand is for extremely reliable computer systems. Secondly, the decrease in hardware costs and the greater sophistication of hardware has created a demand for much more complex and sophisticated software to exploit this.

Software engineering research has resulted in a number of techniques, methods and tools which are intended to improve the way in which software is produced. Many of these doubtless have a beneficial effect when they are used. However, the cost of using them can prevent their use. These costs arise because staff are not trained in the appropriate techniques, or because using them increases the length of the development process, or because they involve investment in new hardware or software tools. The highly competitive nature of the software industry has meant that these short-term costs have been rejected, with little regard for the long-term costs incurred as a result. One such long-term cost is that for product support and maintenance, since errors found at these later stages in the development process are much more difficult and expensive to rectify.

Even when techniques for planning, designing, costing and structuring the software product are used, the emphasis is on debugging and testing as the way to produce reliable software. As a result, insufficient time is given to software design, particularly the exploration of alternative designs, because of the need for an early start to implementation. Much of the design that is done uses informal methods, relying on diagrams or natural language descriptions to communicate the meaning of the specification between the designers and to the implementers. The lack of a precise semantics for such descriptions makes it difficult to reason about the correctness or completeness of the specification, thereby giving greater scope for ambiguity, inconsistencies, errors and omissions. Such errors, introduced at the design stage, are generally harder to remedy when they are found, since they can be fundamental to the entire design and tend to be discovered late in the development cycle.

For all of the reasons given above, software engineering techniques which address the early stages of development are of particular interest. Two such

techniques are formally specifying the product and prototyping the product early in the design process. We introduce these techniques after looking at the additional problems found in designing software for human-computer interaction.

1.2 Software for human-computer interaction

Human-computer interaction (hci) is a relatively new discipline in computing science. Although some of the issues with which it is concerned arose with the advent of interactive teletype devices [Orr 68] [Meadow 70], it is only in the last few years that human-computer interaction has emerged as a discipline in its own right. Evidence for this can be seen in the increasing number of workshops and conferences devoted to the subject [Guedj et al 80] [Gaithersburg 82] [Degano & Sandewall 83] [CHI 83] [INTERACT 84] [CHI 85] [HCI 85] [CHI 86] [HCI 86].

It is not simply an academic interest either. The computer industry, too, is taking human-computer interaction seriously [Thomas 82] [Bewley et al 83] [Alvey 84a] [Reid 85] [Shackel 86]. There are a number of reasons for this explosion of interest in the subject.

One factor is cheaper and more powerful technology, such as high-resolution screens and speech processing, which makes more sophisticated interfaces possible. Another is the recognition that ergonomics, or human factors, can be applied to software as well as to hardware. With businesses, hospitals and defence installations (to quote the examples given earlier) ever more dependent on ever more complex interactive computer systems, it has become important to develop systems that are not only reliable but also offer interfaces which lead to correct use of the systems, being easy to use, resistant to errors and so on.

The primary factor, however, is the proliferation of computer systems, particularly of small, low-cost systems, which has resulted in a much larger and more heterogeneous population of computer users [Moran 81b]. In the early days of interactive computing, software was used either by computer scientists who were familiar with the terminology and tolerant of poor interfaces or by dp professionals

who had no choice but to adapt to the interfaces provided. With the wider use of computers have come new classes of user, such as the "naive" or computer-illiterate user who may well be a professional in some other capacity, or the discretionary user who can choose whether or not to use a system depending on whether it is a help or hindrance to the task in hand. These users require software products that are, amongst other things, easy to learn, easy to use, efficient and robust; in other words, users now have much higher expectations of the user interface to software products.

As stated earlier, though, it is not easy to design good interactive software [Underwood 85]. In addition to all the normal problems of software design, such as misinterpreting or not being given user requirements, there are specific difficulties relating to human-computer interaction.

The major problem is the nature of the human partner in the interaction. People cannot be described in precise, mathematical terms and they have highly individual characteristics and preferences. Consequently it is hard to design an interface for communication between a system and such a partner.

Another reason is that human-computer interaction covers such a wide spectrum of issues. Interaction with the user involves several aspects, such as screen layout, human factors and dialogue structure. A considerable range of interface technology is available, permitting use of keyboards, mice, touch-screens, graphics, voice input and so on. The abilities and understanding of the prospective users of the system can vary enormously, and designers may not appreciate the difficulties and expectations of these different groups of users [Hammond et al 83]. There are also different styles of interaction, such as command-driven, menu-driven or forms-based. Interfaces may be text-based or graphical. The interaction may be under user control, system control or some mixture of the two [Thimbleby 82].

Currently, there is inadequate help available for designers faced with this plethora of decisions to be made. One approach has been to try to establish guidelines for the design of user interfaces. Consequently many authors describe

principles for different aspects of hci design; among them: [Martin 73] [Gilb & Weinberg 77] [Good 81] [Otte 82] [Schneider 82] [Gaines & Shaw 84] [Galitz 85]. However this has not yet yielded a consensus on detailed, useful guidelines for software designers. If anything, the sheer number of guidelines available is more of a hindrance than a help [Gaines & Shaw 86], confusing designers by their quantity and inconsistencies.

The current lack of a theoretical base for deciding what constitutes a "good" user interface means that they are best developed experimentally; in other words, by prototyping the user's interaction with the system [Sime & Coombs 83] [Bury 84] [Norman 84]. From the designers' point of view, this has the added benefit of allowing them to experience the interface themselves, since it can be difficult to visualise a dynamic interaction from a static specification. Moreover the complexity of many interfaces, especially when error handling and on-line help are taken into account, indicates that a formal notation may help interface developers to reason about their designs.

The combination of these factors means than an approach which brings together formal specification with some form of prototyping is likely to meet the particular needs of designers of interactive systems, particularly if the approach can be consistently used for the entire system. The next two sections introduce these two techniques.

1.3 Formal specification

Naur [Naur 82] defines formalism as being expressed "purely by means of symbols given a specialised meaning". Usually mathematics is taken as the basis for the symbols, or notation, used. Formal methods for software specification, then, are usually based on the use of mathematical concepts in describing the requirements on the software. Formal specifications concentrate on defining what is to be achieved by the software rather than how it is to be achieved, although they may also suggest the architecture of the implementation.

The use of a formal mathematical notation gives a precise meaning to the specification, namely that of the mathematics used. This alone has significant advantages for software designers. Given designers familiar with the notation, mathematical notation should be easier to reason about than informal notations, which in turn should enable earlier detection of errors and inconsistencies. A mathematical notation suffers none of the ambiguities of natural language thus allowing precise communication between designers. Finally, its precision makes it possible to perform syntax and type checking of the specification automatically.

Such a specification can be used in a number of ways, depending on the specification language and method being used. The act of constructing a specification can be a major benefit itself, with its attendant comprehension of the problem, concern for consistency and completeness, and documentation of mutual understanding among designers [Guttag et al 82] [Duce & Fielding 84]. However, most advocates of the technique do not stop at this stage.

One approach is to repeatedly transform it mathematically until it is executable as an efficient program [Darlington 81, 85] [Feather 82]. A second way is to use it as the standard against which the implemented program is judged, either formally (by correctness proofs) or informally. Another way is to formulate questions about the desired behaviour of the product (as in "what happens if ...?") and answer them from the specification [Guttag & Horning 80]. Examining the behaviour of the specified system can also be achieved by executing the specification itself. This may be by symbolic execution [Cohen et al 82], or as a prototype of the software [Tavendale 85], or both [Kemmerer 85]. Either method gives valuable feedback to the designers about their specification.

1.4 Rapid prototyping

Prototyping of a software product is seen as a useful software engineering technique in its own right. An analogy with other engineering disciplines can be drawn, whereby a scale model or prototype of a system is often built as part of the

early design phases of development, either to clarify the requirements of the user or to explore alternative ways of meeting those requirements. Consequently, "rapid prototyping" is becoming more and more common, both in academic literature and in industrial practice (see, for example, [Gomaa & Scott 81] [Wasserman & Shewmake 82] [Boehm et al 84] [Berry & Wing 85]).

The traditional software development lifecycle consists of (at least) the following stages as shown in Fig.1.1:

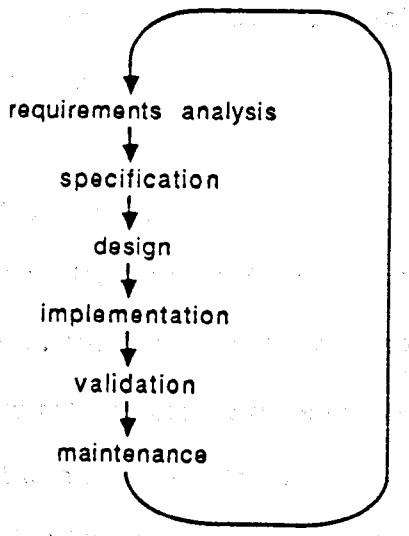


Fig.1.1 Software lifecycle

This has a number of limitations: it does not easily handle areas of uncertainty in requirements or design, the user has little opportunity to influence development until product delivery, and design faults may not be detected until a late stage in the process.

Introducing prototyping into the early part of this cycle has a number of advantages [Dearnley & Mayhew 83]. Prototypes can help clarify and correct user requirements, which may only become apparent to the users themselves as they experience a working prototype. They are often easier to comprehend than either a concise formal specification or an imprecise informal description. They allow dynamic reviews of the design among the design team, and experimentation with alternative designs. They can be submitted to extensive user trials for monitoring and feedback. Users may also find them valuable as a training aid in preparation for the real system.

Prototyping can be seen as complementary to formal specification. Indeed [Swartout & Balzer 82] views the two as being inextricably linked. If users are unclear about their product requirements, it can be difficult to give a formal specification of the design, and a prototype may be the best way to establish the actual requirements [Gomaa 83]. Formal specifications are still subject to errors and a prototype can be used to help designers detect them, especially errors of omission or poor usability. The two techniques are particularly closely linked when the formal specification language is directly executable, so that the specification acts as the prototype [Henderson 86].

1.5 Functional specification languages

One way to gain the benefits of both formal specification and rapid prototyping is to use a formal notation which can be executed directly. A number of notations possess both these attributes; we have chosen to use a purely functional language to achieve executable specifications.

Functional programming languages have a relatively long history in computing science with Lisp as the main (albeit impure) example dating from 1960 [McCarthy 60]. Functional programming, though, was brought to prominence in Backus' Turing lecture [Backus 78], in which he identified the shortcomings of conventional von Neumann computer architectures and languages, and proposed an alternative functional style of programming to overcome the deficiencies.

Conventional languages, however simple or complex, elegant or obscure, are based on the concept of making changes to a global state by means of some form of assignment statement. All other facilities in such languages can be seen as ways of controlling the use of assignment (by constructs like loops or case selections), of limiting the scope for assignment errors (for example, by strong typing) or of organising the structure of parts of the global state (as with data structuring facilities). This style of programming is mathematically complex in theory and has proved unreliable in practice.

In its place, the use of functional languages offers a mathematically-based, high-level approach to programming. Functional languages are built on the mathematical foundations of the lambda calculus and permit clear, concise program descriptions constructed from functions free from side-effects. They continue the trend away from a preoccupation with efficiency towards increased power, clarity and succinctness in programming languages.

This very high level of description together with their mathematical foundations mean that functional languages can be used as a formal notation for specifying software systems. In addition, since the notation is an executable programming language, the specification can be executed as a prototype of the system.

1.6 Terminology

Various descriptions are used by different authors to refer to aspects of human-computer interaction, with many giving specialised (and different) meanings to terms such as "conversation", "dialogue" or "interaction". In this thesis, we refer to the overall communication between the software system and its user as a dialogue. A dialogue is made up of a number of interaction points, or interactions, which are the points in the dialogue where information is communicated from one partner to the other - a single input and/or a single output. Each interaction is specified by an event. An event may also describe other activities in the system (such as changes to the system state) as well as interactions. The order in which these events occur is called the structure of the dialogue.

This thesis presents languages for specifying a dialogue by specifying each event and the order in which they occur.

1.7 Summary

The motivation for this work has been succinctly stated by R.F. Sproull:

> "As the complexity of user interfaces increases rapidly, it becomes increasingly necessary to apply some discipline to the programming of the user interface. ... It should be possible to define a structure for a user interface that helps to organise and simplify its construction and modification."
>
> [Sproull 83, p.135]

We have seen that it is difficult to design good software, that is, software that is reliable, usable and maintainable. In particular, human-computer interaction is an area in which good software design is both crucial and difficult to achieve. Formal specification and rapid prototyping have been advocated as software engineering techniques which help in designing software. This thesis shows a way of applying them to human-computer interaction as a discipline and structure for creating user interfaces.

Interaction makes particular demands of the techniques. A specification should be clear and comprehensible, and should communicate the intentions of the designers. In specifying dialogue, a major feature to be communicated is the sequence of interactions between user and system. Consequently, any notation for specifying dialogue should be able to convey such information.

This work began with an existing formal specification notation and method (me too) which was to be extended to allow specification of human-computer interaction. Several prototype systems were built and used to explore different approaches. The first borrowed notation from CSP ("Communicating Sequential Processes") [Hoare 78] to specify communication between the user and the system. The second was state-based, employing a cycle of actions on that state, where each cycle represents a single interaction in the dialogue. Experience with these methods revealed their complementary strengths and weaknesses, and underlined the difficulty of using a single notation to specify both the structure of a dialogue (ie. the events which constitute it) and the activity involved in each event.

Accordingly, the two approaches have been united in one method with two languages. The first abstracts from the details of the user-system interactions; it specifies the structure of the dialogue in terms of the events that make up the

dialogue and when they occur using a CSP-based notation. The second defines each event, specifying the state transformations which occur for that event.

The combination yields a specification and prototyping method for human-computer interaction, using an event-driven dialogue manager based on CSP. The method has been demonstrated on a number of examples, some of which are described in this thesis. It is based on the me too method of development (described in the next chapter) and the final prototype of the tool embeds its event specification language in me too. However, for production use, the tool was also built using C under UNIX*. This version embeds the event language in C.

The remainder of the thesis is as follows. Chapter 2 surveys other work in formal specification and rapid prototyping, both in general terms and as it relates to human-computer interaction. Chapter 3 introduces the concepts involved in our method of dialogue specification by describing the early prototypes since the concepts were developed during these experiments. Chapter 4 presents the dialogue specification languages. Chapter 5 describes the system that implements them. Chapter 6 highlights implementation issues for the system, and chapter 7 presents conclusions and suggests avenues of further research.

* UNIX is a trademark of AT&T Bell Laboratories

CHAPTER 2

RELATED WORK

This chapter surveys work both in human-computer interaction and in the software engineering techniques which we have chosen to use.

For formal specification, some representative methods are described to illustrate the various approaches possible. The second technique, rapid prototyping, is far from being a generally-agreed method, with many different definitions and techniques. Consequently, we discuss these differences and some of the reasons for them as well as some ways in which prototyping is achieved.

The second part of the chapter describes concepts from the field of human-computer interaction which are relevant to our work before surveying different methods for specifying and prototyping dialogues. Finally, we summarise the contributions made by the work reported in this thesis.

2.1 Software engineering techniques

2.1.1 Formal specification

This section looks at some languages and methods used in the formal specification of software. Two general approaches may be distinguished, namely axiomatic and model-based, although there is some convergence of the two [Horning 85].

An axiomatic, or "property-based", approach defines the properties of operations by giving equations relating them, as in the ubiquitous stack example:

```
pop (new-stack() ) = error
pop (push(element, stack) ) = element
```

where new-stack and push are the "constructor" operations for the stack type, and are considered primitive. In practice, other operations are defined in terms of the constructors. This is the approach taken primarily by algebraic specification techniques (eg. [Guttag & Horning 78] [Goguen & Tardo 79]). One much-quoted application area for algebraic methods has been the description of abstract data

types, although the methods can also be used for more general problems [Goguen & Meseguer 82]. A different axiomatic approach is the use of logic (such as first-order predicate logic or temporal logic) to specify software [Manna & Pneuli 81] [Kowalski 85] [Moszkowski 86].

The model-oriented, or constructive, approach attributes meaning to a specification on the basis of an underlying model whose semantics are already defined mathematically. A data type, or object, is specified by constructing it from basic types whose properties are already known (eg. sets). Its operations are specified in terms of their effect on the object.

The rest of this section describes a selection of individual methods which illustrate aspects of formal specification.

OBJ

One specification language based on algebraic methods is OBJ [Goguen & Tardo 79]. OBJ allows the user to define abstract "objects", where an object may describe an abstract data type (that is, a class of values together with operations which manipulate such values) or an algorithm. Objects can import objects that have already been defined, forming a dependency hierarchy of objects. Operations belonging to an object are specified by algebraic equations as described above. Objects may be parameterised, so that, for example, the object LIST defined for elements of any sort can be instantiated as LIST-OF-INT, specific to integers. This allows libraries of useful objects to be set up.

As a specification language, then, OBJ offers the benefits of formal specification, abstraction and modularity. In addition, it is possible to execute an OBJ specification by regarding the equations as rewrite rules. In order to do this the equations must satisfy two conditions: firstly, that there are no infinite sequences of rewrites; and secondly, that the final result is independent of the order in which the rules are applied. (OBJ provides a facility to overcome the problem of non-termination, by saving the intermediate results of rewrites and prohibiting any

rule which would produce a result that had already been obtained.) OBJ rewrites a supplied term until no further rules can be applied and then returns the resulting term.

More recently, OBJ has been modified and extended to allow it to be used as a very-high-level programming language [Goguen 84]. In consequence, OBJ can be used for both specification and prototyping in the design of software.

VDM

VDM [Bjorner & Jones 82] [Jones 80, 86] is a development method which uses a model-oriented specification language, although it does not exclude axiomatic specifications. In a VDM specification, operations are defined as acting on a state which is constructed from mathematical objects (sets, lists, maps and so on). The mathematical objects used can be manipulated by the operations normally associated with them, for example, set union or list concatenation. The effect of an operation is specified implicitly by a post-condition, which is a predicate relating the input state to the output state. This specifies a class of implementations for the operation with no restriction on the algorithm to be used, save that it satisfy the post-condition.

VDM is not simply a notation, however. It is a method, a way of developing software rigorously, defined by Jones as being "precise without being completely formal". It is iterative in nature, with each iteration moving from the specification towards a more concrete representation, and ultimately to the program itself.

The initial specification is abstract, defining data in terms of mathematical objects and defining operations implicitly. At each iteration, the data types may be made more concrete by choosing a less abstract representation ("reification") and respecifying the operations accordingly, or the operations may be developed towards the implementation ("decomposition"). In either case, the designer has to demonstrate that the result of the iteration meets the specification from which it was derived. Once this has been done, the result may be used as the specification

for the next iteration. This process continues until a satisfactory implementation is reached. The resulting program will, if the method is followed, have been shown to be correct with respect to the original specification.

MIRANDA

If OBJ is an example of the progression of a specification language towards a very-high-level programming language, then MIRANDA [Turner 85] illustrates the opposite development. A purely functional language based on the use of recursion equations, it shares the main characteristics of functional languages - it is a static, definitional language, it has a powerful data-structuring capability, it treats functions as first-class objects (ie. it is higher-order) and it has a mathematical foundation (functions, lambda-calculus and recursion equations). These features are very similar to those required of a specification language.

With the addition of a notation for set abstraction [Turner 82] and a strong polymorphic type discipline, it is not surprising to find MIRANDA being advocated as a language suitable for specification. It has the added benefit that the resulting specifications can usually be executed, so providing a rapid prototyping facility. The equational language used is amenable to mathematical manipulation; in particular, a specification can often be transformed to a more efficient form (or, if the original was not executable, to an executable form), and properties of the specification can be studied by stating and proving theorems about it.

me too

Like VDM, me too [Henderson & Minkowitz 86] [Henderson 86] adopts a model-oriented approach and encompasses both a method and a notation. Like MIRANDA, it is based on a purely functional language, in this case either LispKit [Henderson 80] or a subset of muLisp [muLisp 83]. Like OBJ, me too takes an object-based view of software, requiring the specification of objects (data structures) and operations which act on those objects.

me too proposes an iterative method of software design. The first step (called the Model step) in each cycle is to describe the abstract objects and operations. This is done informally, the result being a list of objects and operations, their intended meaning or purpose described in English, and the functionality of each operation.

The second step (the Specify step) involves giving a formal representation for each object and a formal specification for each operation. As in VDM, objects are constructed from basic mathematical types, in this case, sets, relations, finite functions (maps), tuples and sequences. The operations are defined in terms of the natural operations for the underlying mathematical types. Unlike VDM, however, the operations are explicitly specified, defining how their results can be constructed.

This constructive specification leads to the third (Prototype) step in the method. The specification notation is transliterated into a functional style suitable for execution by a prototyping shell run on LispKit. This form of the specification can then be executed as a prototype. Exercising the prototype is likely to reveal errors, inconsistencies and omissions in the design, which is why the process is iterative, allowing changes to the model and/or its specification.

2.1.2 Rapid prototyping

The growing interest in this technique is evidenced by two major workshops devoted to the subject [Squires 82] [Budde et al 84]. There is still considerable debate over the role, content, validity, costs and benefits of software prototypes [Floyd 84]. Some of this arises from the different concepts covered by the term, so we begin by reviewing the various definitions in the literature.

"Rapid" does not refer to execution time, but is generally taken to mean "quick to produce" since the cost must not be excessive and the prototype is required early in the design process if it is to have any influence in further design. The iterative method of

Design -> Prototype -> Review

implies that the prototype must also be "quick to change" in order to respond to the feedback obtained from earlier versions. Such iteration in the design process is acknowledged to be a valuable mechanism for finding and correcting design errors at an early stage in software development [Sommerville 82] [Bonet & Kung 84] [Berry & Wing 85].

"Prototype" has two very different interpretations - it can mean a "mock-up" [Gregory 84] or "scenario" [Mason & Carey 83] which is a surface presentation of the product usually concentrating on the user interface with little or no functionality behind it. Alternatively, it can mean "bread-boarding" [Botting 85] or "scale modelling" [Weiser 82] which provides the functionality and structure of the logical design with minimal concern for its presentation. What both these interpretations have in common, however, is the desire to clarify and/or explore areas of uncertainty in the requirements and design [Davis 82] and to communicate the decisions made. Prototyping is intended to allow designers to investigate the least certain or most critical parts of a design, be that its user interface or its functionality. Some work has been aimed at prototyping both the interface and the functionality, as in the USE system [Wasserman 86].

The different way in which prototypes can be used accounts for much of the debate over the technique. For example, one argument concerns the fate of the prototype [Patton 83]. Should it be discarded once used [Brooks 75] or should it be developed into the final product [Blum 83]? As pointed out in [Strand & Jones 82], this can depend on the circumstances of the development. In small-scale systems, evolution from prototype to product may be dictated by economic necessity.

More generally, differences arise from that fact that producing a prototype of a system involves limiting that system in some way. The choice of limitation, and hence the kind of prototype, depends on the objectives for the prototype [Smith 82]. The designers may limit the performance, perhaps to allow the use of an expressive but inefficient language for prototyping. They may decide to limit the scope, for example, to cover only simple cases or to use small amounts of data. Or they may

limit the functionality by simply omitting some aspects of the system, such as error handling.

How prototyping is actually achieved is another issue [Taylor & Standish 82]. As has been shown by the examples in §2.1.1, rapid prototyping is often closely related to formal specification methods, particularly where the specification can be executed. When this is possible, it arises from the use of a mathematically-based notation which, being precise and unambiguous, can be "understood" by some form of interpreter [Goguen & Meseguer 82] [Stavely 82] [Belkhouche & Urban 84] [Kowalski 85] [Lee & Sluizer 85].

There are other, less mathematical, approaches to rapid prototyping. They may still involve a formal description, in the sense of Naur's definition, where the notation is an existing programming language [Boehm et al 84] or a specially-developed language [Mason & Carey 83]. Existing software may be re-used, or a prototype may be produced rapidly using ordinary software development techniques by restricting the functionality.

Surveys of various techniques used in rapid prototyping have been compiled in [Carey & Mason 83] and [Hekmatpour & Ince 86b].

2.2 Human-computer interaction

Although some attempts have been made to apply the general specification techniques described above to human-computer interaction (hci), most work has involved the development of specialised notations. This section describes both specialised and general techniques used to specify user interfaces. Before doing so, however, various concepts have to be introduced.

2.2.1 Overview of concepts

Separation

Many authors agree that components dealing with interaction should be separated from components forming the actual application, eg. [Casey & Dasarathy 82]

[Feldman & Rogers 82] [Huckle & Bull 84] [Williges 84]. One or two point out that in practice such a separation may be difficult to define or achieve [Reid 85] [Cockton 86]. However, when it is possible, the benefits of dividing the software into these subsystems are considerable.

An analogy is with database management systems; there, data storage, organisation and retrieval are delegated to a single "back-end" subsystem, the DBMS. The application programmer no longer has to handle a substantial part of the overall system, thus simplifying the task in hand. Similarly, interaction between system and user accounts for a large part of any interactive system [Sutton & Sprague 78]. Accordingly, it seems reasonable to create a single front-end subsystem, called the "user interface management system" (or UIMS) [Kasik 82] [Buxton et al 83] [Pfaff 85]. With such a system available, the application designer is concerned only with the functionality of the application and the interface designer with the dialogue. As a result, we can envisage employing specialist designers to deal with these different aspects of the system [Johnson & Hartson 82] [Norman 84].

Separating the application from the interface handling is thus a means of simplifying the designers' task, but it brings other benefits as well. Given this separation, it should be straightforward to provide different interface components for an application (eg. menus, forms or commands) or to ensure the same style of interface across many different applications. An example of the latter approach is the COUSIN system which gives a consistent form-based interface to several different applications [Hayes 85]. In a distributed environment, a further advantage is that dialogue components can be located in distributed workstations. If the components are adapted to take account of individual user characteristics, the user interface at each workstation can be tailored to its user(s) [Carey 84].

Layers of interaction

Human-computer interaction can be subdivided, corresponding to different layers of the interaction. The number of layers involved depends on which model of

interaction is adopted [Moran 81a] [Foley & van Dam 82] [Neilsen 86]. The most widespread approach follows a linguistic model, seeking to define the dialogue at each of the lexical, syntactic and semantic layers.

Within this model, the interface contains two languages - user input and system response. The individual keystrokes, button pushes or the like that make up the "words" of the user input are defined as tokens in the lexical layer, as are the components and characteristics of any system output (such as colour, position, choice of window). The syntactic layer defines the sequence of input and output. In particular for user input, it defines valid "sentences" in the user language, ie. the allowable combinations of tokens which the user may input. The effects of such sentences are defined by the semantic layer, which specifies the functionality underlying the interface. Dividing the user interface into these different aspects simplifies its design by allowing different parts of the problem to be considered separately.

The semantic layer is defined by the application designer in the course of designing the objects and operations of the application. User input is translated into invocations of the operations supplied by the application system. System responses are also specified by operations within the application. This layer does not form part of a UIMS since it is defined by specifying the application. This can be done using any of the existing formal specification techniques, such as algebraic or model-based methods. Although the semantic operations are not specified as part of the interface, they will, of course, be used in the interface specification.

Secondly, we consider the syntactic layer, which defines the sequence of inputs and outputs in the dialogue. It specifies more than two individual sequences for input and for output, though. It has to specify the relationship between the two languages, the "interaction logic" [Strubbe 85]. It is thus the key to specifying the dialogue, since it draws together all the various aspects of the dialogue. This layer defines what we have called the dialogue structure. In a UIMS, it is handled by a component called the "dialogue manager". This thesis is primarily concerned with

specifying this layer and providing a dialogue manager to allow it to be prototyped.

Finally, the lexical layer defines the layout of the screen, windows, colour, internal representation for user input and so on. In the past, this area has received the most attention when the user interface has been considered - how tokens are to be made up ("a HELP button, or the user typing HELP?", "what form of command abbreviation?", ...) and screen presentation ("what are the friendliest colours?", "how much information can be displayed on the screen?", ...) and so on, ad infinitum. In some ways it is not surprising that this should be so, since this surface detail is very apparent to the user and is often the first cause for complaints. However, this part of the UIMS has usually been regarded as straightforward, performing relatively simple transformations of input from and output to the user [Edmonds 82]. More recent investigations involving graphics systems indicate the correct handling of feedback to the user is rather less trivial than first anticipated [Kamran 85] [Olsen et al 85]. However, since we have chosen to concentrate on the syntactic layer, we assume the existence of a "presentation manager" within the UIMS which is responsible for this layer.

With the components mentioned above, the architecture of a UIMS may be pictured as in Fig.2.1.

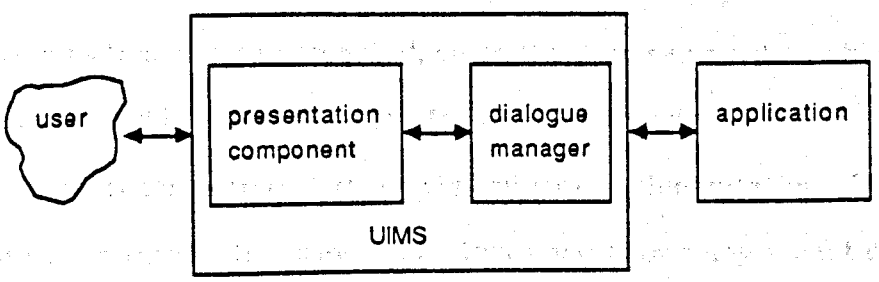


Fig.2.1 Structure of a UIMS

It should be noted that most authors extend or elaborate upon the layers outlined above. All those cited above add a conceptual layer to describe the user's model of the tasks to be peformed by the system. Moran and Nielsen elaborate the layers by further subdividing their concerns. For example, Moran introduces the

"interaction level" which incorporates some aspects of dialogue structure as well as such details as key presses required from the user.

2.2.2 Prototyping techniques

A number of techniques have been devised for prototyping user interfaces. This section describes those which are not directly related to a specification language.

The least formal and least expensive approach is to use fixed screen displays to demonstrate what the interface will look like at key points in the dialogue. This may be entirely manual, using printed diagrams, or may involve the use of an animation system to run through a sequence of screens, with or without user input [Mason & Carey 83]. Although it provides a concrete display to the user and is quick to set up, it is also restricted in what can be shown to or experienced by the user.

Another method is to substitute the designer for parts of the interface that are missing [Good et al 84] [Kelley 85]. The user interacts with the system as intended in the production version, but for missing facilities or unrecognised input, the input is sent to a second terminal where the designer responds as required. This allows a more realistic experience of the system and also enables the designer to see how the user would like to be able to use it. However it may be restricted in the kinds of systems that can be presented, since the designer must be able to respond in a realistic time, and is not appropriate for large-scale trials with many users.

A third approach is to undertake preliminary implementation. This is less restrictive than either of the previous methods and allows any aspect of the user interface to be included. It can be expensive both to produce and modify such implementations, however.

Consequently, methods which allow a realistic prototype to be derived automatically from a description of the interface are much in demand. Commercially, this has led to fourth-generation tools such as QuickBuild [ICL 86] which allow entire interactive application systems to be generated very quickly.

Such tools tend to be founded on database technology and are usually restricted to commercial dp systems. More generally, a number of methods are directly linked to a specification language which is used as the basis for the prototype, either driving an interpreter or being used to generate the prototype itself. Since they are so closely related to their specification technique, they are described in the next section where appropriate.

2.2.3 Specification techniques

This section surveys the various techniques proposed as means of specifying human-computer interaction. Drawing on the linguistic model, the most frequently-used techniques are those derived from traditional language specification and analysis, namely BNF (Backus-Naur Form) and state transition networks; these are described first.

BNF

In BNF, the grammar defines the input language. The terminals of the grammar are taken to represent primitive user input actions, while non-terminals group and structure these actions. As it stands, BNF is not sufficient to describe the syntactic layer of dialogues since it has no means of relating valid input to its effects, including outputs. However, it can be augmented by adding notation to describe actions to be taken when a phrase or sentence of the input language is recognised. This approach is adopted, for example, in the compiler-compiler YACC [Johnson 78], as well as in specifying the user interface [Lawson et al 78].

An example of a BNF specification for a simple logging-on dialogue, extended by actions as above, might be:

```
<logon>      ::= LOGON (1) <user-id>
<user-id>    ::= <bad-user>* <good-user>
<bad-user>   ::= %USER (2)
<good-user>  ::= %USER (3)
```

where

```
(1) output:      "user name?"
(2) condition:   not REGISTERED-USER(%USER)
    output:      "invalid user name, try again"
(3) condition:   REGISTERED-USER(%USER)
```

[Reisner 83] extends BNF in a different way to include the cognitive actions of the user as well. The resulting specification allows estimation of user performance with the interface prior to its implementation. This enables the designer to experiment with alternative interfaces without having to implement them. [Payne & Green 83] also presents a variant of BNF which is used for analysing specifications of command languages.

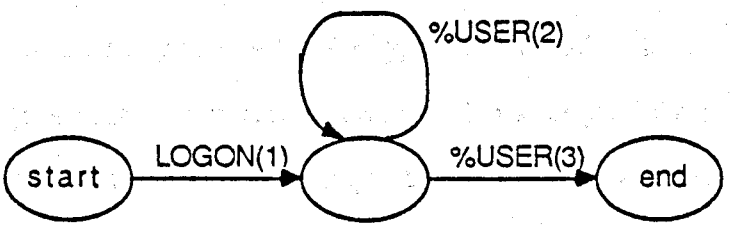
However, as suggested earlier, it is often helpful for the designer to experience the interface personally [Lieberman 83] and to allow users to evaluate it [Damodaran & Eason 83]. Consequently the formality of BNF is exploited as a means of generating the user interface. Prototyping is achieved by providing an interpreter for BNF which allows the designer to generate the user interface components directly from the specification [Hanau & Lenorovitz 80] [Olsen & Dempsey 83].

Note that these grammar-based approaches are centred on user input, with output regarded as an "action". [Shneiderman 82b] moves away from this asymmetric treatment of input and output by using a "multi-party grammar". This labels the source of each utterance, either the user or the system. [Bleser & Foley 83] presents a specification language based on this idea. The language is capable of describing the three basic layers of the interaction described above. For the syntactic layer, it describes the grammar of both input and output, and their relative sequencing. They use this notation to define interface characteristics and to analyse the resulting specification with respect to the various guidelines given for human-computer interaction.

State transition networks

When state transition networks are used, the specification describes a set of states (nodes) and transitions between them (directed arcs), either in a text form or diagrammatically [Parnas 69]. Transitions are labelled with user inputs. A transition from a state will be traversed if its input is given while the system is in

that state. To mimic the use of non-terminals in BNF and to reduce the size of individual specifications, the label on a transition may name a sub-network [Denert 77]. Also, as with BNF, the original concept is extended to allow actions and/or outputs to be associated with transitions [Woods 70]. The example dialogue given above could be specified (using the same actions) as:



[Jacob 85] relates this approach to the interaction layers described in [Foley & van Dam 82]. The semantic layer is defined by the operations supplied by the application. The syntax and lexical layers are described in separate groups of diagrams, with lexical sub-diagrams being "called" from syntax diagrams. Each transition can have various attributes attached to it: a condition to be satisfied, an input to be matched, an output to be generated, an action to be taken or the name of a sub-diagram to be called. The diagrams, in their text form, are executable by an interpreter, thus, as with BNF, the interface component can be created directly from the specification.

SYNICS [Edmonds & Guest 84] takes a similar approach, though it uses BNF to define the user input language whereas Jacob uses state transition networks throughout. It also permits input to be taken from the application, thus allowing the same analytic techniques to be used for system responses as for user input. SYNICS makes it explicit that the interaction is seen as a collection of dialogue events. A dialogue event corresponds to a named state together with its outgoing arcs and other attributes. [Guest 82] found that designers preferred the state-transition-network version of SYNICS to an earlier version based on BNF, partly because of the natural way in which the former could express the sequence of events in a dialogue.

[Alty & Brooks 85] describes CONNECT which, like SYNICS, associates the

attributes for each step with the nodes rather than the arcs of the network. It identifies different types of node to handle various activities, so that a "task" node does not communicate with the user but with the task required. A major aim of this work is to investigate ways of dynamically reconfiguring the network in order to provide dialogues which can be adapted to the needs of users as they engage in them.

An alternative method of allowing different styles of interaction to suit different classes of user is proposed in [Hagglund & Tibell 83]. Instead of describing a data structure (the state transition network) with a single interpreter, so that different dialogue styles depend on changes to the data structure, they provide a state-transition-network data structure which can be used by several different interpreters. Each interpreter implements a different style of dialogue control (eg. menus or forms). All the information needed for the interaction is represented as attributes of the states. The attributes include prompts, defaults, lexical information and transition instructions. They are the union of the slightly different attribute sets needed for the different styles.

This does not cover all of the state-transition-network-based techniques - since it is a popular approach there is a sizable amount of the literature on the subject. [Jacob 83] gives a comprehensive survey of the use of state transition networks (and BNF) for specifying human-computer interaction. More recent papers include [Wartik & Pyster 83] [Ho 84] [Olsen 84] [Kieras & Polson 85] and [Wasserman 85].

Interaction events [Benbasat & Wand 84]

An interaction event is defined as a point in the dialogue where user input is required. A dialogue is then a sequence of basic interaction events, and the specification of a dialogue consists of event definitions together with a specification of the flow of control. Like a SYNICS dialogue event, each interaction event is made up of common generic elements, some of which are mandatory (eg. prompts or the flow of control) while others are optional (eg. help or input checks). The

sequence of events is specified by the "flow control" element in each event; that is, it is defined within the events themselves. Constructs available are based on the usual programming structures: sequence, choice and iteration.

In this example, we omit some of the elements and compress the notation for the sake of space:

Event-id	Prompt	Check	Flow-control
COMMAND	"?"	RESP ∈ Valid-Cmds	(RESP=LOGON) -> LOG
LOG	"user name?"	REGISTERED-USER(RESP); TEXT="invalid user..."	....

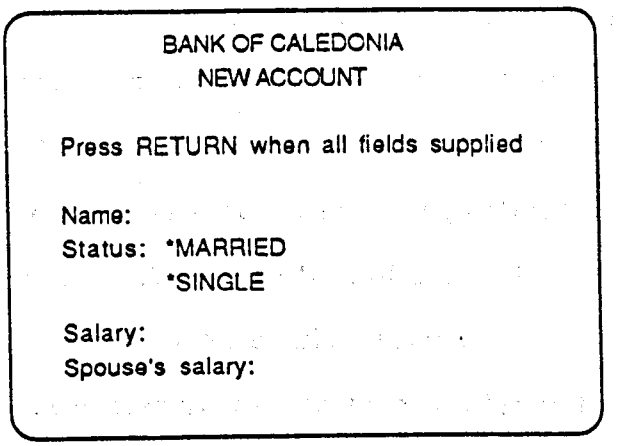
This method does not make use of the layers of interaction described earlier; all information about the dialogue is contained in the one description. In particular, it does not enforce a syntactic/semantic split, as the code implementing actions or conditions is incorporated into the dialogue description itself. The method is primarily designed to handle command languages at present; it has no facilities for screen definition.

Frame-based techniques

This approach is often found in systems for computer-aided learning (CAL), and aims to provide an "authoring language" with which the author of a CAL lesson can describe the dialogue of the lesson. The "frame" used here is not related to the AI notion introduced in [Minsky 75] but is merely a unit of information for display. Like an interaction event, it packages together all aspects relating to that step in the dialogue. [Barker 84] describes a dialogue programming language, MICROTEXT, in which each frame contains many of the attributes of interaction events, such as flow control, input checks, help information and so on. MICROTEXT, though, is screen-based and so it also allows screen positions and layout to be defined.

[Lafuente & Gries 78] takes a slightly different frame-based approach. The language separates the sequencing of frames (the flow control) from the representation of the frames. This allows the frames to be entirely declarative in nature, with the imperative sequencing handled by the (Pascal) program in which the

frame descriptions are embedded. In addition to the usual ability to define the content and layout of items in a frame, the designer can incorporate behaviour rules in a frame description. Thus a frame description defines the items contained in it but also relationships between them and rules governing their behaviour. As an example, consider one frame which describes the "new account" display for a banking application:



The rules for this frame include

```
require salary, name;
require card(status) <= 1;
require salary2 if MARRIED in status and salary < 10000
let salary2.display be (MARRIED in status);
terminate if RETURN-KEY;
```

These rules, for example, end the frame when RETURN is pressed, will not allow the frame to be sent without entering a salary, only display the spouse's salary field when the applicant is married and only require that information under certain conditions.

Object-oriented techniques

A number of techniques use an object-oriented approach. [Jacob 86] defines an interface using interaction objects, each of which can specify its dialogue with the user as well as its data and procedures. The dialogue specification itself may be local or inherited. The multiple-inheritance mechanism means that a significant economy in specification can be achieved, since generic objects can be specified and then reused by more specialised objects.

[Cook 86] is also concerned with providing generic interfaces using the object-oriented approach. Instead of state transition networks, though, the entire interface is specified in a non-executable functional language. The paper shows how the specification of a simple window manager can be specialised to handle windows of different types.

[Smith et al 84] use an object-oriented programming language with special facilities for interface handling. The system is broken down into individual tasks, within a hierarchic structure. A task is defined in terms of its attributes, which are called slots, eg. the code, expansion or port slots. Sequencing is described by the module hierarchy and by any (partial) ordering defined within the module. The port slot defines a primitive step in the flow of data between tasks. The interaction style used (eg. menu or command-driven) is not specified explicitly in the task description but is selected by the interface handler on the basis of information in the description.

A predecessor of the object-oriented approach was the artificial intelligence concept of frames [Minsky 75], which included the concept of inheritance within a hierarchy of frames. GUS [Bobrow et al 77] explored their use for controlling a dialogue. More recently, [Sandewall 82] defines command languages for office information systems in terms of frames, capitalising on the similarities between the languages, eg. create an X, delete an X, print an X, where X may represent such objects as a mail message, an appointment or a room booking.

[Lieberman 85] has a more limited field of interest, namely menu systems, but shows how object-oriented techniques can simplify the specification of such systems.

Knowledge-based techniques

Even before the popularity of expert systems, knowledge-based techniques were advocated for specifying hci [Hopgood & Duce 80] [Durrett & Stimmel 82]. Inputs are received in working-storage and production rules are held in long-term storage (the "knowledge base" in expert system terminology). The interaction

handler compares the contents of the working-storage with the conditions in the production rules. When a match is made, the actions of the matched rules are triggered; this may cause an output, some internal action and/or the addition of further items to working-storage. The matched items are removed from working-storage and then the process is repeated.

[Kieras & Polson 85] combine the use of production rules with state transition networks. The rules describe the user's view of the task to be performed, based on the GOMS model [Card et al 83]. The behaviour of the system is defined by state transition networks. Their interest is in analysing the complexity of both views of a system and the correspondence between them.

Hopgood & Duce note that rule-based methods can lead to simplicity and economy of description (particularly when simultaneous user inputs are permitted), avoid specifying order unnecessarily on the sequence of user inputs, and create the possibility of adaptive dialogues (by dynamic modification of the knowledge base). These properties are being investigated in an Alvey project, "Adaptive Intelligent Dialogues" [Alvey 84b] [Durham 85].

Other specialised techniques for hci

Input-output tools propose a hierarchic method of specifying interaction [van den Bos et al 83]. An input-output tool is a named object with an input rule, an output rule, internal tool definitions and a tool body. The input rule is analogous to a production rule in a grammar; it specifies the input pattern capable of triggering the tool. The rule can name other tools (as non-terminals) and has operators for selection, interleaving, repetition and sequencing. If the rule can be matched to the user input, the tool body is executed and output is generated. With the addition of prefix (guard) functions on input rules and post-test functions on output parameters, the descriptive power of input-output tools is extended up to that of context-sensitive grammars.

The Descartes system [Shaw et al 83] is based on principles of language design, such as the provision of suitable abstractions. The designer specifies individual

elements of the screen display and can then compose them using generic rules. For example,

for COMPOSE use BACKGROUND=white, FORMAT=framed

Dialogue: COMPOSE of Command-area, Help-area
with ALIGN=vertical

Help-area: SCROLL of [PROGRAM-VAR of HelpText]
... etc

The interaction is managed by an application-specific module which can be generated, in part at least, from the specification.

General formal specification methods

[Anderson 85] uses general formal specification techniques (the algebraic language CLEAR, together with regular expressions, context-free grammars and a denotational style) for interaction rather than developing a specialised notation. Armed with this array of techniques, he is able to state mathematical formulations of desirable properties of user interfaces and demonstrate whether or not they hold for a specified interface. [Mallgren 83] presents event algebras as an extension to an algebraic specification method to handle interaction, primarily to enable the formal definition of input/output primitives in interactive graphics languages. Another extension to algebraic specification is proposed in [Chi 85], using the flow expressions described in [Shaw 80].

The model-based method is also being investigated for specifying hci. A significant specification, in that it is of a reasonable size and has been implemented, is for a display editor [Sufrin 82]. This was defined using Z, a model-based notation that can specify operations either axiomatically or constructively. Other proposals involve VDM. Of these, the most ambitious project is EPROS [Hekmatpour & Ince 86a], which seeks to integrate all aspects of software development in a single framework. For dialogues, EPROS uses state transition networks. A primary goal of EPROS has been to ensure that at each stage of development from specification to implementation the system, both functionality and user interface, is executable.

Functional languages are also addressing the issues of synchronisation and

input/output directly [Abramsky & Sykes 85] [While 86], but have not yet been developed sufficiently to offer a clear way to specify hci. Of these, a promising approach is that advocated by While, in which a language based on temporal logic describes constraints on the execution of Hope programs [Bailey 85]. [Feldman 82] presents a syntax-directed approach to interface specification, using the functional language FP to give meaning to the syntactic constructs of the input language for a line editor.

Techniques for concurrent input

With the advent of multiple interaction devices, such as touch-screen, mouse and keyboard, has come the need to be able to specify concurrent input and the ability to omit specifying ordering of the inputs where appropriate. This section groups a number of methods by their effect (allowing concurrency) rather than by the technique employed.

One notation is flow expressions [Shaw 80], which is an extension of regular expressions to provide a notation for describing graphics command languages. The extensions allow the notation to handle interleaving of symbols, cyclic activities and synchronisation. [Chi 85] proposes using flow expressions together with algebraic specification to describe user interfaces, although how they are to be combined is not made clear in his paper.

Based on CSP and CCS, squeak [Cardelli & Pike 85] is a programming language developed primarily to address the issue of concurrency among interaction devices. Although it is mainly concerned with the device level, it can be used to describe higher-level dialogues such as the logon example. However, the example given here illustrates a particular feature of squeak; its explicit handling of time. This example is a process describing the detection of button presses on a mouse:

```
Click = DN? . ( wait[clickTime] .
                ( UP? . click! . Click )
                || ( down! . UP? . up! . Click ) )
```

The || construct specifies alternative options dependent on a timeout. The form of

this is

wait[x](a.P) || Q

which means that P will occur if event a happens within time x. If event a does not happen in that time, the process continues as Q. In this example, then, when the button is pressed (event DN occurs), the process waits for it to be released (event UP). If it occurs within the "clickTime" specified, a non-primitive "click" event is generated. If not, a non-primitive "down" event shows that the button is being held down. In this case, when the UP event is received, the "up" event is sent.

A third approach is taken by [ten Hagen & Derksen 85], which describes dialogue components using dialogue cells. A dialogue is made up of steps, each defined by a dialogue cell. A cell is built from four basic elements: user action, external system reaction (echo), internal state changes, and conditions determining when the action occurs. To allow parallel input, cells may be active simultaneously and the user may input to any active cell.

Statecharts [Harel 86] are a graphical, extended version of state transition networks. They define states graphically, as nested boxes, and trigger transitions by events. Statecharts offer a number of features not found in the more usual state-transition notations. For example, they permit specification of default entry points to enclosed states, history-dependent defaults, and concurrent processing. The charts give no detail as to the effect of events, however; this is specified separately.

In [Jacob 86], the state-transition-network technique is extended to handle modern "direct manipulation" interaction techniques [Shneiderman 82a]. Jacob adopts the object-oriented paradigm and represents items on the screen by interaction objects. Each object has an associated transition diagram specifying how the user may interact with the object. An executive activates and suspends the individual dialogues as co-routines, calling each only when an appropriate token is available for it, so that the user can switch between interacting with the various objects displayed on the screen.

2.2.3 Features of the techniques

This section picks out some of the main features found in the techniques described above.

Defining dialogue steps

Most of the techniques described are based, either implicitly or explicitly, on breaking the dialogue down into primitive steps, each with a common structure. These dialogue steps have various mandatory and optional attributes, which together describe the characteristics of that step. The sequencing of the steps may be included in these attributes or may be specified separately. Often this decomposition of a dialogue is specified in some form of data structure. The interaction can then be animated by one or more processors which interpret the data structure representing the dialogue.

Combining dialogue specifications

Clearly, where a dialogue is decomposed into its constituent parts, some mechanism for creating the overall structure from the parts must be provided. In some techniques, this is achieved by a hierarchic structuring of the parts (eg. dialogue cells, input-output tools); in others, by explicit command (eg. "COMPOSE" in Descartes); in others, by the nature of the specification (eg. state transition networks where node and arc descriptions are all part of the one notation).

Offering alternative interaction styles

Interactive systems are used by various classes of user, ranging from "novice" through "casual" up to "expert", with different methods of interaction being appropriate for each [Badre 84]. For example, a menu-driven system may be suitable for novice or casual users but can be tedious for experts. Consequently, it is useful to be able to offer different dialogue styles for any given application.

One approach is simply to specify the different dialogues individually and then rely on the separation of interface from application to allow the appropriate

interface module to be used with the application. Alternatively, the interface components can be altered dynamically as the dialogue progresses (as in CONNECT). [Hagglund & Tibell 83] proposes a sufficiently generic dialogue structure that can be processed by more than one interpreter, each offering a different interaction style. The system described in [Smith et al 84] bases the choice of style on the properties of the information required rather than on the user.

Prototyping interaction

Rapid prototyping of software is increasingly acceptable as part of the software development process. It is particularly appropriate when designing interactive software since it allows the interface designers to see a dynamic presentation of the interface. For example, [Sufrin 82] notes that some design decisions were only made after experimenting with alternative implementations of the display editor.

Where the notation used for specification is executable, the specification itself acts as the prototype. Many of the techniques described in this paper enable the interface component to be prototyped from the specification, eg. BNF with actions, state transition networks, interaction events, input-output tools.

2.3 Contributions of this work

The languages, tool and method presented in this thesis (collectively known as SPI – for specifying and prototyping interaction) bring together many of the desirable features currently scattered across the techniques described in the previous section.

SPI sets out the dialogue in terms of discrete events acting on a state, clearly separating the structure of the dialogue from the effects of the individual events within that structure. The overall structure is specified in a subset of CSP [Hoare 85]. This defines the order of events, the possible sequences of events that can occur in a dialogue. Using CSP notation allows concurrent and partially-ordered

input to be specified. The events themselves are simply me too operations which specify when an event can occur and what happens to the state when it does. Such events are defined in a second notation, designed as a shorthand for the me too that would otherwise have to be written.

The SPI languages have been embedded in me too, and, like me too, they are formal, declarative and executable. Embedded in C, the event specification language provides better performance and more scope as a production-quality tool.

SPI adopts the me too method for software design: an iterative, prototyping activity based on formally specifying the behaviour of the system. In addition, it offers the transition from specification to implementation. A SPI specification can be reworked in the programming language C and the result can be executed as the implementation of the dialogue.

Using CSP as a way of controlling me too operations (in the form of events) is similar to proposals for using temporal logic to control execution of Hope equations [While 86]. Thus, although not the main intent of SPI, the languages offer a way of handling synchronisation in a functional language. In particular, SPI allows input/output in a functional language, as the synchronisation between program and user.

Overall, SPI demonstrates a way in which formal specification and rapid prototyping can be applied to human-computer interaction in order to reap the benefits of using such techniques in this increasingly important area.

CHAPTER 3

EARLY PROTOTYPES OF SPI

This chapter describes the background to the present system. It traces the development from the original me too language by means of three prototyped systems which were used to explore different ways of characterising human-computer interaction. As such, it serves both to introduce the concepts involved in the SPI architecture and languages and to illustrate the general method of software development advocated in me too.

In functional languages, a natural approach is to consider an interactive function as mapping an input stream to an output stream. Although adequate for demonstration purposes, this view becomes overly complex for more realistic systems. This observation led to the first group of prototypes of SPI which investigated the use of a more succinct notation for streams. The notation was based on CSP channels and did make it easier to read and write specifications of dialogues. However the notation, although useful, was restricted in a number of ways, especially for specifications of larger systems.

As a result, a simpler characterisation of interaction was developed. This second technique, called ECS, moves away from explicit stream handling, using instead state transformation and interpretation. While this method, not surprisingly, clarifies the state transformations involved, it tends to obscure the structure of the dialogue.

The third set of experiments returned to CSP for modelling a dialogue, but no longer using a stream-based implementation. In this notation, a dialogue is specified as sequences of events, where each event represents an interaction or other activity occurring in the dialogue. This proved to be an excellent way of outlining the structure of a dialogue and led to the architecture and languages presented in this thesis.

The sections that follow describe the functional approach originally used in

me too, then each of the three exploratory systems. me too notation is explained as it is introduced, but for reference Appendix 1 gives a fuller description of the language.

3.1 Streams in me too

Since me too is a functional specification language, a natural starting point in describing dialogues is to experiment with existing methods used in functional programming. The usual approach is to view the description of an interactive system as a function from its input stream to its output stream [Henderson 82], and to assume that constructing this description is a straightforward task.

Employing this technique may well be adequate for small functional programs, but experience has shown that for more realistic programs it results in specifications that are difficult to read and to reason about. me too has been used to specify a variety of interactive applications, including spreadsheets, expert system shells [Jones et al 85] [Bruce 86] and a decision support tool [Minkowitz 86]. In the process of developing these applications, it has become apparent that dealing with input and output by this traditional method can become complex and certainly obscures the meaning of the specification, particularly where a function deals with more than one input or output stream [Jones 84]. Since the aims of formal specification include comprehensibility [Liskov & Zilles 75] and better communication between designers [Henderson & Minkowitz 86], such a shortcoming has to be taken seriously by those who advocate the use of functional languages to specify software.

By way of an example, the "logon" example used in chapter 2 is given here using me too. The operations specifying the dialogue assume the existence of a table of users and passwords and appropriate underlying operations for checking them. These constitute the "application", the semantics of the dialogue, and are specified in Appendix 2.

The structure of the operations is determined by the way in which interactive

me too prototypes are executed. A prototyping shell for me too, called ProtoKit, has been built at Stirling. It enables a designer to execute a me too specification, creating and manipulating the specified objects using the specified operations. For an interactive operation, that is, one which maps an input stream (the keyboard) to an output stream (the screen), ProtoKit provides the "run" command. This expects the interactive operation to be of a particular type, namely:

(in -> (out x user-state x in))

For future reference, we will call this the type runnable-process. The operation supplied to "run", therefore, is expected to take an input stream and map that into an output stream, some form of result and the unused portion of the input stream. In practice, interactive operations usually have functionality.

(user-state -> (in -> (out x user-state x in)))

or, abbreviated:

(user-state -> runnable-process)

This is to allow use of predefined objects as the user-state. In the logon example, "run" is called with the operation "cmd-level(udb)" as its parameter, where the dialogue is specified by the following me too operation:

```
cmd-level(udb)(kb) ≡
   letrec (s1,d1,k1) =
          let cmd = head(kb)
              kb = tail(kb)
          in
             if cmd="logon" then logon(udb)(kb)
             else if cmd= ....
             else letrec (s2,d2,k2) = cmd-level(udb)(kb)
                  in list(cons(errmsg1,s2),d2,k2)
   in list(cons("?",s1),d1,k1)
```

To understand this specification, it is first necessary to appreciate the way in which "run" drives the interactive operation supplied to it. The "run" command seeks to display each item in the output stream as it becomes available, and all other activity in the runnable-process (input and/or computation) is only undertaken in order to extract the next output item.

In me too, the let and letrec expressions introduce local declarations within an

operation. Here letrec is used to identify the individual components returned by a runnable-process.

The first step in this operation is to add the prompt "?" to the output stream, so that "run" will display it. Next, "run" seeks to evaluate the rest of the output stream, here identified as "s1", part of the result returned by the rest of the operation. In order to obtain the next output item, "run" has to evaluate this inner expression, so it begins by removing the next item from the input stream, referring to it locally as "cmd". This input is used to determine the result of the operation. If a valid logon command has been given, the result is created by calling another operation "logon" (also of type runnable-process) with what remains of the input stream. If no valid command has been given, an error message is added to the output stream and the rest of the result is obtained by recursively calling "cmd-level".

Thus this operation prompts the user for a command and processes the reply, either calling a further operation or giving an error message before starting again. The other operations needed to specify this dialogue have a very similar structure, and so are given below without further explanation.

```
logon(udb)(kb) ≡
   letrec (s1,d1,k1) =
          let user = head(kb)
              kb = tail(kb)
          in
             if registered(udb,user)
             then pwd(udb,user)(kb)
             else letrec (s2,d2,k2) = logon(udb)(kb)
                  in list(cons(errmsg2,s2),d2,k2)
   in list(cons("user:",s1),d1,k1)

pwd(udb,user)(kb) ≡
   letrec (s1,d1,k1) =
          let pass = head(kb)
              kb = tail(kb)
          in
             if validpwd(udb,user,pass)
             then shell(user)(kb)
             else letrec (s2,d2,k2) = logon(udb)(kb)
                  in list(cons(errmsg3,s2),d2,k2)
   in list(cons("password:",s1),d1,k1)
```

These three operations, which specify just the beginning of an interactive

session, all have to deal with the input and output streams explicitly, adding and removing elements as appropriate and then handing streams on the next operation or returning them to the previous one. While the specification can be understood, it requires some effort because the explicit stream handling obscures the meaning of the operations. This lack of clarity is compounded when multiple input and output streams are used.

However, the fact that the same structures are repeatedly used in taking input and constructing output indicated that it would be possible to provide some form of shorthand notation which could be translated into these structures. This development is described in the next section.

3.2 Introduction to streamCSP

Recognising that a suitable notation can be a powerful means of communicating ideas [Iverson 79], the first prototyped system investigated the use of a notation to express the stream-handling characteristics of interactive operations. Clearly, this system went through a number of iterations, but here we present only the latest version.

The notation used stems from the language of "communicating sequential processes" (CSP) which was introduced by Hoare [Hoare 78] to provide both a simple way of describing input to and output from a program (or process) and a means of achieving concurrency of execution among processes. Here, notation based on a subset of CSP is used for the description of input and output in deterministic systems [Henderson 84].

It should be made clear at the outset that the CSP introduced in this section differs in a number of significant ways from that given by Hoare. In order to distinguish it from the original CSP, it is referred to as "streamCSP". Before discussing the differences, however, we give the notation used.

3.2.1 StreamCSP notation

In streamCSP, a process P is defined by a process expression:

p = <process-expr>

where a process expression is defined as follows:

if P and P' are process expressions, B and B' are boolean expressions, Q and Q' are processes, then the following are also process expressions

c ? v → P	input – on channel c, receive a value into v and do P
c ! e → P	output – put value of e on channel c and do P
return(x1,...xk)	termination ("skip") – naming components of process state to be returned
(B → P [] B' → P' [] ...)	conditional ("alternative") – if B then P, if B' then P', ...
Q(x1,...,xk)	process invocation – call Q with the named components of the process state
Q;Q'(x1,...,xk)	sequential composition – call Q with the process state supplied; when it terminates, call Q' with the process state returned by Q

Note that streamCSP uses channels for communication. In the original version of CSP [Hoare 78], channels were not present; instead processes were uniquely named and communication was between named processes, eg.

R = (P!e → Q?x → R)

which sends data to process P and receives data from process Q. However the disadvantages of this approach led to the introduction of named channels for communication [Hoare 83, 85]. StreamCSP adopts this later development and names input and output channels rather than source and destination processes.

3.2.2 Departures from CSP

In this section we assume a CSP with channels as the basis of comparison.

Firstly, CSP uses unbuffered channels whereas streamCSP, which implements each channel by a lazy infinite list, has buffered channels of potentially infinite

capacity. Each system can model the other, since CSP can specify buffered channels by using buffer processes and streamCSP can specify systems with unbuffered channels by establishing some form of hand-shaking protocol between the processes involved. Nevertheless, this is a significant semantic difference between the two notations.

Secondly, employing lazy lists as input channels precludes the use of input guards in alternative commands (here referred to as conditional process expressions). In CSP, the presence or absence of input on a channel can be used to choose between alternative actions, as in

```
( ch1 ? x → P
[] ch2 ? y → Q )
```

where the choice between P and Q depends on which of the channels (ch1 or ch2) receives an input value first. In streamCSP, lazy evaluation of an input channel means that it cannot be checked to see if a value is present or not - ProtoKit will simply wait until an input appears on the first channel it checks (of course, input may not appear at all on that channel). Consequently the guards in a conditional process expression are restricted to being of boolean type.

A third difference is in the treatment of non-deterministic choice between guards. In CSP, non-determinism is introduced in the conditional process expression by allowing more than one condition to evaluate to "true" and by not defining the order of evaluation. Non-determinism is avoided in streamCSP by guaranteeing the order of evaluation of conditions. Even if several conditions may be true, the choice of which will be used is determined by the order of evaluation, the order being that in which the conditions are specified. Consequently the meaning of

```
(B → P [] B' → P' .... )
```

is

```
if B then P
else if B' then P'
...
```

Analogously

```
if B then P else P"
```

expresses

```
( B → P [] true → P" )
```

The two notations are exactly equivalent in streamCSP.

It can be seen from this discussion that our aim was not an accurate implementation of CSP but its use as a convenient notation for stream manipulation. The following sections show how this notation has been used and implemented.

3.2.3 Using streamCSP notation

As a first example, consider a decision table application. The application supplies various operations to interrogate and manipulate a decision table of the form shown in Fig.3.1 (specified in Appendix 3). With these operations, we can specify a dialogue in which the system asks the user questions from the table until a decision is reached.

		Responses			
Questions	Q1	y	y	n	n
	Q2	y	n	y	n
Decisions		a	b	b	c

Fig.3.1 Decision table example

This can be specified by the process

```
dts(dt) = ( is-decision(dt) → scr!<"decision:",dt>
                  → return(dt)
          [] true → scr!<question(dt),"?"> → kb?ans
                  → dts(prune(dt,question(dt),ans))
          )
```

In this specification, the process first decides whether or not the decision has been reached by calling the application operation "is-decision". If so, the decision is the remaining tree "dt" and is output to the screen on channel "scr", whereupon the process terminates. If more information is needed to make a decision, the user is asked the next question by sending it to the screen. The user response is accepted from the keyboard channel "kb" into a local variables "ans". This response is then used to change the current version of the table (in the "prune" operation supplied by the application) and this new version is handed to a new call of the "dts" process.

Note the use of sequence construction <...> to create a single item of output text from a number of constituent items. A sample dialogue with this system (with the user input underlined) might be:

```
Q1? y
Q2? y
decision: a
```

Alternatively, we could specify a system which allows the user to choose the order in which the questions are answered:

```
dtu(dt) = ( is-decision(dt) → scr!<"decision:",dt>
                → return(dt)
          [] true → scr!"?" → kb?q&a
                → dtu(prune(dt,get-q(q&a),get-a(q&a)))
          )
```

A sample dialogue with this system might be

```
? Q2 y
? Q1 n
decision: b
```

In all these examples, we use "kb" to denote the keyboard input channel and "scr" for the screen output channel.

The earlier "logon" example can be re-specified as

```
process cmd-level(udb) =
        scr!"?" → kb?cmd →
          ( cmd="logon" → logon(udb)
          [] cmd=...
          [] true → scr!errmsg1 → cmd-level(udb) )

process logon(udb) =
        scr!"user:" → kb?user →
          ( registered(udb,user) → pwd(udb,user)
          [] true → scr!errmsg2 → logon(udb) )

process pwd(udb,user) =
        scr!"password:" → kb?pass →
          ( validpwd(udb,user,pass) → shell(user)
          [] true → scr!errmsg3 → logon(udb) )
```

This description is much more succinct, readable and comprehensible than the earlier version with explicit stream manipulation. StreamCSP has been used to specify a variety of dialogue styles (described in detail in [Alexander 85]) and has proved effective in communicating the structure of interactive operations. Before discussing its advantages and limitations, though, we outline its implementation.

3.2.4 Implementing streamCSP

StreamCSP is implemented as a language embedded in me too. StreamCSP (in its functional S-expression form) is translated by a preprocessor into standard me too which can then be run as a prototype. The preprocessor systematically rewrites terms in the source notation using a set of rewrite rules until no more rules can be applied [Finn 84]. Thus streamCSP is defined by a set of rewrite rules which translate processes into stream-handling me too operations of type runnable-process. The rules for the version of streamCSP described above (which assumes a single input channel – the keyboard – and a single output channel – the screen – both implicitly named by the preprocessors) are based on rules devised by S.B. Jones of the University of Stirling, and are given in Appendix 4.

3.2.5 Evaluation of streamCSP

Compared with the explicit stream manipulation required in me too, streamCSP was a significant step forward in making it easier to specify and prototype interactive systems using a functional specification langauge. One reason for its appeal is that it turns out to be well-suited to convey the sequence of events in a dialogue. Since the structure of human-computer interaction is primarily a sequence of exchanges between the user and the system, a notation which clearly sets out that sequence makes the specification easier to understand.

Unlike most other dialogue specification notations, streamCSP does not decompose a dialogue into its primitive steps, each with certain predefined attributes, such as a condition or an action. Instead it specifies a dialogue as being made up of one or more processes. A process is not restricted to specifying a single exchange between user and system, but usually specifies a group of related interactions.

The version of streamCSP described in this section was only one of a number of prototypes used to explore this approach to dialogue specification. This particular version has been presented because it offers the clearest way of introducing the concepts involved, rather than demonstrating how far streamCSP

can be taken. Other ideas which were investigated included alternative ways of composing processes and use of multiple input and output channels. Each of these developments added significantly to the complexity of specifications given in streamCSP and thus began to expose the limitations of this approach.

A major deficiency in streamCSP is that, while it simplifies specifications of simple dialogues, it too becomes unwieldy and complex when more powerful constructions are required. Secondly, streamCSP displays an inherently imperative form. This is undoubtedly useful for setting out the structure of the dialogue, but is not considered a desirable feature of a functionally-based specification language. A further problem is the substantial way in which it differs from Hoare's CSP since this can contribute to misunderstanding of streamCSP specifications.

While streamCSP did offer an improvement in its ability to specify interactive me too operations, the disadvantages given above encouraged investigation of a different method.

3.3 Edit-Compute-Show (ECS) paradigm

In searching for an alternative view of dialogue using a functional specification language an obvious question to ask is how Lisp, the archetypal functional language (even if not purely functional), achieves its highly interactive capability.

The answer lies in its use of a very simple form of interaction: the Read-Eval-Print loop. Thus, the Lisp system is in some state, with some functions defined, perhaps, and some data available. It accepts an input from the user ("Read"), evaluates that against the state ("Eval") and gives some appropriate response ("Print"). In the process of evaluating the input, the state itself may be changed.

This approach models a finite-state machine [Minsky 72], where both the new value of the state and the output depend only on the current input and the current value of the state. In fact, finite-state machines (FSMs) are related to stream-processing as well: a process which receives an input stream and the current

state and returns an output stream and a new value of that state is a function with internal state, modelling a FSM [Sheeran 84].

The Read-Eval-Print paradigm, then, was adopted as a different way to specify dialogues. It does not involve any additional language features in me too; instead it prescribes a way of constructing me too operations.

3.3.1 Introduction to ECS

ECS employs the notion of the "state" of the dialogue advocated by the proponents of transition networks. The system is in a particular state until given some user input. It reacts to that input by transferring to a new state and perhaps producing some output.

Thus the current status of the dialogue is modelled by a state. Possible actions on that state are edit (to change it by providing the input), compute (constructing any output and the new state) and show (making the output available to the user). These actions, after some initialisation, are repeatedly executed in a cycle as shown in Fig.3.2. The name of the technique is derived from this cycle of events.

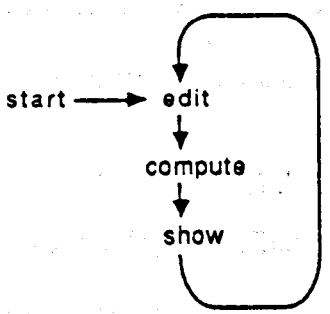


Fig.3.2 Edit-Compute-Show cycle

Operations for "edit" and "show" are supplied as standard me too operations, together with a further me too operation which executes the ECS cycle and acts as the interpreter of the state. This interpreter is specified in Appendix 5.

The designer's task, then, becomes one of specifying the operation(s) needed to implement each step of the dialogue, ie. supporting the "compute" action. These operations have to specify a number of tasks to do with recording the new state of the dialogue (such as creating output, requesting input or indicating the end of the

dialogue). They can then be executed by the ECS-interpreter, thus providing a prototype of the dialogue.

The state is represented as a "finite function" (or "map", in VDM terminology). This can be thought of as the table given in Fig.3.3.

index	entry contains
INPUT	user input
INPUT-REQD	boolean flag for input
OUTPUT	created by "compute"
TERMINATE	boolean flag to stop
DB	application database

Fig.3.3 ECS state

The properties of this table are:

- all rows are optional and are unordered
- the set of indices for all rows present in the table (ie. the domain of the table) is given by
  dom(state)
- the presence of a row is indicated by the presence of its index in the domain of the table, which can be checked by the usual set membership test
- state ds {i} removes row i (domain subtract)
- state ⊕ {i→v} overwrites row i with value v
- the contents of row i are accessed by
  state[i]def
  returning the default given by "def" if the row is not in the table

More formally, it is defined as a me too object:

ECS-state = ff(Index,Contents)

where

Index = { INPUT, OUTPUT, DB, TERMINATE, INPUT-REQD }
Contents = Text U Boolean U AppState
Text = seq(Atom)

In particular, for indices INPUT and OUTPUT, the contents are Text; for indices INPUT-REQD and TERMINATE, the contents are Boolean; and for DB, the contents have the application-specific type AppState.

The operations on the ECS-state are

start : AppState -> ECS-state
edit : ECS-state x Text -> ECS-state
compute : ECS-state -> ECS-state
show : ECS-state -> Text

Of these, the standard operations edit and show are very simple, just adding input text to the state or extracting output text:

```
edit (st,i) ≡ st ⊕ { INPUT→i}

show (st) ≡ st[OUTPUT]<>
```

The start operation is supplied as part of the specification. Its purpose is to set up the dialogue state as required by the compute operation. This may simply be to include the application-state, as in

```
start(as) = { DB→as }
```

but it may also initialise other parts of the state if required (since the compute operations can extend the state to save and use local information).

3.3.2 Using ECS

First we specify the decision table example introduced in §3.2.3. The me too notation used here is that for finite functions, as given above. The operations needed for the example are:

```
start(dt) ≡ { DB→dt }

compute(st) ≡ dts(st)

dts(st) ≡
   let inp = st[INPUT]<>
       dt = st[DB]
       st' = st ds { INPUT }
   in
   st' ⊕
     if is-decision(dt)
     then { TERMINATE→true,
            OUTPUT→ <"decision:",dt> }
     else if INPUT ∉ dom(st)
     then { INPUT-REQD→true,
            OUTPUT→question(dt) }
     else { DB→prune(dt,question(dt),inp) }
```

The alternative interaction style where the user controls the order of questions is specified by:

```
compute(st) ≡ dtu(st)

dtu(st) ≡
   let inp = st[INPUT]<>
       dt = st[DB]
       st' = st ds { INPUT }
   in
   st' ⊕
```

```
if is-decision(dt)
then { TERMINATE→true,
       OUTPUT→ <"decision:",dt> }
else if INPUT ∉ dom(st)
then  { INPUT-REQD→true, OUTPUT→"?" }
else   { DB→prune(dt,get-q(inp),get-a(inp)) }
```

Unlike streamCSP, the availability of input is signalled to a compute operation by the presence of the INPUT entry. The absence of input causes the operation to prompt for it. The operations in this example remove input from the state as soon as it is used but in other situations it might be appropriate for the input to remain available for more than one ECS-cycle.

For comparison with streamCSP, we also give a specification of the logon dialogue. This example reveals the major problem with ECS, namely the way in which a natural sequence of interactions has to be controlled explicitly by the designer.

```
start(udb) ≡ { DB→udb, NEEDS→CMD }

compute(st) ≡
    let needs = st[NEEDS]CMD
    in
        if needs = CMD then cmd-level(st)
        else if needs = USERNM then logon(st)
        else if needs = PWD then pwd(st)
        else st ⊕ { OUTPUT→"error" }

cmd-level(st) ≡
    let inp = st[INPUT]<>
        st' = st ds { INPUT }
    in
        st' ⊕
              if INPUT ∉ dom(st)
              then { INPUT-REQD→true, OUTPUT→"?" }
              else if inp = "logon"
              then { NEEDS→USERNM }
              else { OUTPUT→"error: bad command" }

logon(st) ≡
    let inp = st[INPUT]<>
        udb = st[DB]
        st' = st ds { INPUT }
    in
        st' ⊕
              if INPUT ∉ dom(st)
              then { INPUT-REQD→true, OUTPUT→"user:" }
              else if registered(udb,inp)
              then {NEEDS→PASSWD, USER→inp }
              else { OUTPUT→"error: bad user" }
```

```
pwd(st) ≡
    let inp = st[INPUT]<>
        udb = st[DB]
        user = st[USER]<>
        st' = st ds { INPUT }

    in
    st' ⊕
        if INPUT ∉ dom(st)
        then { INPUT-REQD→true, OUTPUT→"password:" }
        else if validpwd(udb,user,inp)
        then { OUTPUT→"logon completed", NEEDS→SHELL-CMD}
        else { OUTPUT→"error: bad password",
               NEEDS→USERNM }
```

3.3.3 Implementing ECS

Since ECS involves no additional language constructs in me too, all that is required is to implement the standard operations for the edit and show operations in the ECS-cycle (which were given in §3.2) and the ECS-state interpreter.

The ECS-interpreter runs through the cycle, giving output if there is any and ending when the TERMINATE flag is set to "true". Informally, its behaviour is:

- stop if the TERMINATE flag is set
- request input (the "edit" step) if INPUT-REQD set
- execute the "compute" operation
- display output (the "show" step) if there is any

It is specified formally in Appendix 5, using me too to handle the single input stream and output stream explicitly. Since this is the only place where input/output occurs and the interpreter is provided as part of ECS, the dialogue designer is not required to deal with input/output streams directly.

3.3.4 Evaluation of ECS

ECS provides a data-based, declarative way of specifying the structure of a dialogue. It constructs a dialogue from a number of steps, each corresponding to one edit-compute-show cycle. The activities involved in each step are determined by processing the ECS-state data structure. This data structure and its processing are relatively simple, yet the method is capable of describing the same wide variety of dialogues as streamCSP [Alexander 85].

In its view of dialogue as consisting of discrete steps, each having the potential to use or alter attributes of the state, ECS has adopted a similar style to many other dialogue specification techniques. A compute operation specifies the possible transformations of the dialogue state, complete with conditions and actions, and so can be seen to be comparable with state transition networks.

However in practice ECS yields cumbersome specifications. Moreover, because each use of the compute operation in a cycle is distinct from the previous use, there is no automatic sequencing of events or actions. The dialogue designer can be forced into specifying these sequences in detail, where streamCSP handles them as a matter of course. These deficiences can make it difficult to determine the structure of the dialogue being specified. Since this is a major requirement for a dialogue specification language, this is a serious flaw in the method.

3.4 Introduction to eventCSP

The ability of streamCSP to express the sequence of interactions and activities in a dialogue and the failure of ECS in this respect led to further experiments with CSP as a notation for dialogue specification.

In more recent years, Hoare's presentation of CSP has moved from its Algol-like origins which used communication between processes as its means of synchronisation [Hoare 78] to a more general event-based approach using events (including communication events) as the synchronisation objects [Hoare 82b, 83, 85]. This change in approach prompted the development of a simulator to allow the animation of processes written in this later style of CSP. The availability of this simulator for event-based CSP provoked investigation into its use as a notation for describing interactive systems. This notation, which is a subset of the language presented in [Hoare 85], is referred to as "eventCSP" in the remainder of this thesis.

In using eventCSP, we seek to abstract the structure of the dialogue (represented by events) from the details of the actual inputs, outputs and state transformations that occur. Unlike streamCSP, eventCSP is not merely a notational convenience, but is intended to implement (a subset of) CSP.

3.4.1 EventCSP notation

A process, then, describes the behaviour pattern of some object in terms of events which affect it. In choosing the events considered appropriate for an object, no consideration is given to which are caused by the object (such as an output) and which are caused by its environment (such as an input). As far as the behaviour of an object is concerned they are all events in which it participates in some way, regardless of their origin.

In eventCSP, a process is defined as follows:

if e, e1,...,en are events and P, P1,...,Pn are processes, then
the following are also processes

(e → P)	- (prefix) engage in event e then behave like P
(e1 → P1	- (choice) engage in e1 then behave like P1,
[] e2 → P2	or engage in e2 and behave like P2, etc
[] ...	
[] en → Pn)	
P1 ; P2	- (sequence) P1 followed by P2 if P1 terminates
P1 || P2	- (parallel) P1 in parallel with P2
skip	- successful termination
abort	- no further interaction

The syntax for eventCSP, both abstract and concrete, is given in Appendix 6.

Among these definitions, the parallel operator (||) offers considerable scope for innovation in specifying interaction. For example, we can easily specify "two-handed" (ie. concurrent) input, where the user is free to use, say, a pointing device and keyboard together. Or we can use || to separate and synchronise the activities of related processes. Later examples will illustrate both these uses of the parallel operator.

Unlike streamCSP, both the syntax and the semantics of this notation are closely based on CSP as defined in [Hoare 85].

3.4.2 Using eventCSP

As a first example, this notation is used to describe the behaviour of the decision table dialogues introduced in §3.2.2:

```
dts = ( is-decision → (give-decision → skip)
      [] not-decision → (ask-question → (user-answer → dts)) )
```

This specification outlines the behaviour of the system-driven style of interaction. The process first chooses between the "is-decision" and "not-decision" events. If a decision has been reached, it proceeds to deliver that decision, indicated by the "give-decision" event, and terminates. If not, the events "ask-question" followed by "user-answer" indicate that the next question from the table is asked and a response accepted from the user. The process then continues to behave as "dts", ie. deciding if a decision has been reached and acting accordingly.

For a series of events, as occurs above, the bracketing can be dropped from the specification, so that the other interaction style for decision tables can be specified as

```
dtu = ( is-decision → give-decision → skip
      [] not-decision → prompt → user-reply → dtu )
```

These do not provide detailed specifications of the decision table system, particularly as they rely on the event names to give meaning to the specifications. However each specification does represent the structure of the dialogue more clearly than streamCSP since it is so uncluttered by detail.

Another example is the Click process from [Cardelli & Pike 85] which was given in chapter 2. This process handles the button-pressing activity of a mouse:

```
Click = ( DN → ( UP → click → Click
               [] wait>clickTime → down → UP → up → Click
               ) )
```

DN and UP represent the depression and release of the mouse button. We have dispensed with the special "wait" construct and replaced it by orthodox CSP using an event which is triggered if the time is exceeded. The other events represent the signals sent to the process currently using the mouse.

The next example specifies the logon dialogue:

```
CmdLevel = ( prompt →
               ( logon-cmd → Logon
               [] other → errmsg1 → CmdLevel
               ) )

Logon = ( prompt-for-user →
               ( user-ok → Pwd
               [] not-user-ok → errmsg2 → Logon
               ) )
```

```
Pwd = ( prompt-for-pwd →
                ( pwd-ok → skip
                [] not-pwd-ok → errmsg3 → Logon
                ) )
```

The final example in this section is also taken from [Cardelli & Pike 85] and specifies concurrent input from a mouse and keyboard. First we define a simple process describing the activity of the mouse:

```
Mouse = ( DOWN → get-position → send-position → UP → Mouse )
```

Here the DOWN and UP events represent depressing and releasing the mouse button. When the button is pressed, the current position of the mouse is determined and communicated to the process which is controlling the screen activity.

The process defining keyboard use is also straightforward:

```
Kbd = ( get-char →
          ( newline → send-line → Kbd
          [] text-char → add-to-line → Kbd ) )
```

Characters are accumulated in a line buffer until a newline character is received, then the completed line is made available to the controlling process.

The Mouse and Kbd processes have no events in common so running them in parallel, as in

```
Mouse || Kbd
```

allows interleaved use of the devices, with no constraint on the order of user input. To manage their joint use, we have a controlling process, Text, with which Mouse and Kbd are independently synchronised.

```
Text = ( send-position → save-position → Text
       [] send-line → write-line → Text )
```

Receiving a new position from the mouse has no effect on the screen, although the last position sent is remembered. Receiving a line from the keyboard device causes it to be displayed at the current position. The entire interaction is specified as

```
Text || Mouse || Kbd
```

Although a simple example, this demonstrates the way that concurrent use of interaction devices can be modelled in eventCSP. Modern direct manipulation displays can be specified in a similar way. Each object on the screen is represented

by a process whose initial events determine when that object is selected (eg. by checking the cursor position). The entire display is specified as the parallel operation of the processes for all the objects it contains. When an object is selected, the appropriate process is executed.

3.4.3 Process labelling

A process P labelled by l is denoted

l:P

and each event of P is then labelled with l. A labelled event e is the pair l.e. The process l:P engages in the event l.e whenever P would have engaged in e.

This labelling allows us to make multiple use of processes. Suppose we have two similar front-ends for applications - one for database access and one to access a frames knowledge base:

```
dbfe = (prompt →
            ( stop? → skip
            [] query? → query → dbfe
            [] stats? → statistics → dbfe
            [] update? → update → dbfe
            [] ...
            [] anything → error → dbfe ) )

ffe = (await-input →
            ( end? → skip
            [] list? → listframes → ffe
            [] show? → showframe → ffe
            [] del? → delframe → ffe
            [] ...
            [] other → error → ffe ) )
```

It is useful, given their similar structure, to be able to write a single eventCSP specification of a general front-end process and then reuse it as necessary. Thus, a generic front-end is

```
fe = (prompt →
            ( end? → skip
            [] cmd1? → act1 → fe
            [] cmd2? → act2 → fe
            [] cmd3? → act3 → fe
            [] ...
            [] other → error → fe ) )
```

which allows the specific front-ends to be defined by

```
dbfe = db:fe
ffe = f:fe
```

where "db" and "f" are labels.

3.4.4 Implementing eventCSP

Unlike streamCSP, eventCSP is not a language embedded in me too. Instead, an eventCSP specification is represented by a data structure which is interpreted to provide a simulation of the behaviour of the specified processes. The user running such a simulation acts as the environment for the processes, choosing which event will happen next and observing the results of that choice.

The implementation of operators in eventCSP are loosely based on the implementations given in [Hoare 85]. The primary difference is that processes are not treated as functions (essentially that is the approach taken by streamCSP and [Neely 83]), but as a description of possible event combinations. Details of the simulator, which was originally developed by Peter Henderson of the University of Stirling, will be given in chapter 5.

An example of running the "dts" specification would be

```
> RUN ( call(dts), ... )
PICK ONE OF { is-decision, not-decision }
> PICK(bad-event)
WRONG- PICK ONE OF { is-decision, not-decision }
> PICK(not-decision)
PICK ONE OF { ask-question }
> PICK(ask-question)
PICK ONE OF { user-answer }
> PICK(user-answer)
PICK ONE OF { is-decision, not-decision }
> PICK(is-decision)
PICK ONE OF { give-decision }
> PICK(give-decision)
PICK ONE OF { tick }
> ...
```

where ">" is the system prompt and the "tick" event is offered by the skip process to indicate successful termination.

Although tedious to execute and somewhat unrealistic, this is a useful way of examining the behaviour of processes, and can be considered a valid, if limited, form of prototyping. Despite the limitations, observing this simulated activity in the

system is helpful in designing the system, particularly when the parallel construct is used. Moreover the specification clearly lays out the structure of the intended interactions in a precise, simple and formal way.

3.5 Summary

This chapter has described the early prototypes used to explore methods for specifying and prototyping dialogues between user and system.

Initially, streamCSP gave us a way to encode stream-handling me too operations more succinctly and comprehensibly. The second approach, ECS, had a more traditional view of dialogue as composed of primitive interaction steps and provided a simple model of human-computer interaction. Its deficiences in communicating the dialogue structure took us back to CSP but no longer basing it on stream-handling functions.

It is clear that using eventCSP is not sufficient to formally specify interactive systems, though, since no actual meaning is given to the events. Also, individual selection of each event is not an appropriate method for demonstrating a prototype of an interactive system.

All of these methods offered improvements over me too stream handling and all proved adequate for specifying several styles of dialogue. However for the reasons given in this chapter, none of them could be considered an entirely satisfactory method.

Taken together, though, it can be seen that the methods possess complementary strengths and weaknesses. Both streamCSP and eventCSP clearly set out the structure of the dialogue. ECS, on the other hand, provides a simpler and more declarative model for dialogues and makes the state transformations involved explicit. These various features are desirable in any dialogue specification language, and so it was felt that it was appropriate to develop some synthesis of the methods.

Of the techniques given in this chapter, only eventCSP is retained in its entirety. The notations and implementations for streamCSP and ECS have now been

discarded. However the ECS model for dialogue has been kept and combined with eventCSP, resulting in the method presented in this thesis. The structure of a dialogue is specified using eventCSP. Each event is then separately specified as a state transformation operation with an associated predicate over the state which determines when that event can occur.

The next chapter describes how this combination of eventCSP and state transformations attached to interaction steps is used to specify and prototype dialogues.

CHAPTER 4

DIALOGUE SPECIFICATION USING SPI

The architecture presented in this chapter recognises firstly that CSP (in the form of eventCSP) offers a convenient and expressive description of the structure of a dialogue and secondly that decomposing a dialogue into its primitive steps with associated state transformations is a simple yet powerful model for human-computer interaction. This chapter presents the fourth (and final) prototype in our development of a dialogue specification and prototyping method.

The overall behaviour of the system is specified using a subset of CSP (called "eventCSP") in terms of individual events, each of which defines a single interaction and/or activity in the dialogue. The eventCSP specification can be exercised simply as a simulation of the dialogue or it can be used to control the execution of the events in a prototype of the dialogue. To use it for prototyping dialogue, each event has to be specified, stating any output to be given, any input required and any state transformations to be made. Events are specified in a separate notation, the Interaction Specification Language: eventISL.

In this chapter, first we show how eventCSP can be used for prototyping dialogues by augmenting it with event operations. The eventCSP language has already been described in the previous chapter (§3.4) so it is not given again here. The second section introduces eventISL as a language for specifying the event operations. The remaining sections give further, more substantial, examples to demonstrate the languages.

The details of the implementation of the SPI system which allows the simulation and prototyping of dialogues are deferred to the next chapter.

4.1 Event specification

Given an eventCSP specification, how is it to be extended to allow a more realistic prototype of the dialogue?

From the brief discussion showing the use of the simulator in chapter 3, we can see that two aspects need to be handled. Firstly, there must be some mechanism for selecting the next event that happens without asking the user to make an explicit choice. With decision tables, for example, we need to determine whether or not the user has reached a decision point when choosing between the events "is-decision" and "not-decision". Secondly, having picked an event by some means, that event must be able to cause the appropriate activity in the dialogue before stepping on to the next interaction point in the dialogue. Again with the decision tables example, we would expect the event "give-decision" to output the result to the user.

In order to address these two requirements, we return to the ECS use of the Read-Eval-Print model and of a state which records the current status of the dialogue.

4.1.1 The dialogue state

As in ECS, the dialogue state is a finite function, consisting of a system part with pre-defined entries and an application-specific part containing any objects needed by the application. Informally, it is an extensible table as shown in Fig.4.1. Note that ECS index names have been abbreviated for convenience and postfixed by "$" to distinguish them from any application entries in the state.

index	entry contains
IN$	user input
IR$	boolean flag for input
OUT$	system output
STOP$	boolean flag to stop
DB$	application database at start

Fig.4.1 SPI dialogue state

Formally, it is described as the me too object:

$$DlgState = SysState \times AppState$$

where the system part of the state is as follows

SysState = ff(SysIndex,SysContents)
SysIndex = { IN$, OUT$, DB$, STOP$, IR$ }
SysContents = Text U Boolean U InitAppState
Text = seq(Atom)

In particular, for indices IN$ and OUT$, the contents are Text; for indices IR$ and STOP$, the contents are Boolean; and for DB$, the contents have the application-specific type InitAppState. InitAppState is the type of an initial value supplied for the application. The application can extend the dialogue state and may use information in this initial object to do so. Typically, several values may be recorded in this entry which the application later separates into individual entries in the application-specific part of the state, AppState. This has the type

AppState = ff(AppIndex,AppContents)

where AppIndex and AppContents are dependent on the application involved. The actual types required by a particular specification are explicitly recorded as part of that specification.

4.1.2 Event operations

For each event named in the eventCSP specification, we need to be able to decide when it may be selected and the effect that it has on the dialogue state if it is selected. Consequently, an event is specified in two parts: a guard [Dijkstra 75], or condition, which determines when that event may be selected and an action which describes the state transformations associated with that event. (This contrasts with eventCSP simulation where events are selected by the user and their behaviour is inferred from their names.)

These two aspects of an event can be described using me too operations on the dialogue state. For example, an event which is always ready to prompt with "?" can be specified by the pair of operations:

condition: DlgState -> Boolean
condition(dlg) ≡ true

action: DlgState -> DlgState
action(dlg) ≡ dlg ⊕ { OUT$→"?"}

For each event, then, we require two me too operations - one to specify the

condition, the other to specify the action. These operations are linked to the event by the event name, so that the operations for an event called "is-decision" would be named "is-decision-C" and "is-decision-A" for the condition and action operations respectively. The next section shows how the events in the decision table example can be specified using event operations to extend the eventCSP specification.

4.1.3 Decision table example

This section presents me too specifications of the event operations needed for the decision table example. Some similarity with the ECS approach can be seen but here no attempt is made to describe the structure of the dialogue, since this is specified separately using eventCSP.

As a reminder, we repeat the eventCSP part of the specification:

```
dts = ( is-decision → give-decision → skip
      [] not-decision → ask-question → user-answer → dts )

dtu = ( is-decision → give-decision → skip
      [] not-decision → prompt → user-reply → dtu )
```

In addition to the underlying application operations for decision tables, these event specifications make use of some additional operations on the dialogue state:

```
getdb(dlg) ≡ dlg[DB$]
getinp(dlg) ≡ dlg[IN$]<>
```

The specification assumes that DB$ holds the decision tree and that AppState uses the types:

```
AppIndex = { QU }
AppContents = Question
```

with QU mapped to the current question.

Some of the events are used by both the interaction modes, user-driven and system-driven:

```
is-decision-C(dlg) ≡ is-decision(getdb(dlg))
is-decision-A(dlg) ≡ dlg

not-decision-C(dlg) ≡ not is-decision(getdb(dlg))
not-decision-A(dlg) ≡ dlg

give-decision-C(dlg) ≡ true
give-decision-A(dlg) ≡ dlg ⊕ { OUT$→<"decision:",getdb(dlg)>}
```

```
prompt-C(dlg) ≡ true
prompt-A(dlg) ≡ dlg ⊕ { IR$→true, OUT$→"?" }
```

The user-driven mode requires operations to ask for and process a user response:

```
user-reply-C(dlg) ≡ true
user-reply-A(dlg) ≡
        let dt = getdb(dlg)
            qu = get-q(getinp(dlg))
            ans = get-a(getinp(dlg))
        in
            dlg ⊕ { DB$→prune(dt,qu,ans) }
```

The system-driven mode requires operations for "ask-question" and "user-answer". The first of these extends the application-specific part of the state to record the question that has been asked.

```
ask-question-C(dlg) ≡ true
ask-question-A(dlg) ≡
        letrec dt = getdb(dlg)
               qu = question(dt)
        in
            dlg ⊕ { IR$→true, QU→qu, OUT$→qu }

user-answer-C(dlg) ≡ input-present(dlg)
user-answer-A(dlg) ≡
        let dt = getdb(dlg)
            qu = dlg[QU]O
            ans = getinp(dlg)
        in
            dlg ⊕ { DB$→prune(dt,qu,ans) }
```

Now that a formal meaning has been given to the events in an eventCSP specification, it is time to link the two descriptions together to provide an executable specification for use as a prototype.

4.1.4 Prototyping with event operations

The first step is to incorporate the state used by the eventCSP simulator (referred to as the simulator state from now on) into the dialogue state. The simulator state has not yet been formally defined; we leave this until chapter 5, where the simulator is described, and simply refer to it as having type SimState. A new entry is added to the system part of the dialogue state, which now has the form shown in Fig.4.2.

index	entry contains
IN$	user input
IR$	boolean flag for input
OUT$	system output
STOP$	boolean flag to stop
DB$	application database at start
SS$	simulator state

Fig.4.2 SPI dialogue state (extended)

The me too definition is extended to reflect this:

SysIndex = { IN$, OUT$, DB$, STOP$, IR$, SS$ }
SysContents = Text U Boolean U InitAppState U SimState

with SS$ being mapped to a value of type SimState.

The event simulator can now be used, as before, to offer possible events and to step on in the current process when given an event selection. However, the user is no longer required to make a direct choice of the next event. Instead, this is determined by the condition operations for the possible events. The condition operation for each of the possible events is evaluated and, of those which evaluate to true, one is selected.

Once an event has been selected, its action operation is invoked to perform the required state transformation and the simulator is used to step on to the next point in the eventCSP specification. The event manager which controls all this activity, using the simulator where appropriate, is described in chapter 5.

By providing a formal description of both the structure and the effect of events in a dialogue, we now have a method for formally specifying and prototyping interactive systems. A primary goal, that of clearly setting out the dialogue structure, is achieved using a subset of an established formal notation (CSP). Together with formally specified event operations, the resulting specifications can be executed to give a prototype of the system.

However, describing event operations in me too involves giving much detail which could be generated automatically. Consequently a more concise notation has been developed for specifying these event operations.

4.2 Introduction to eventISL

Events in an eventCSP specification are given meaning by event operations. Instead of specifying these operations directly in me too, now we use a language (eventISL) which allows the operations to be specified more concisely. The constructs in eventISL are translated into me too, in the style shown in the previous section.

EventISL is a language for describing the attributes of events and the state transformations they produce. The current form of eventISL is derived from experience in specifying several dialogues of different styles. It is minimal, in that it offers only what has been found to be necessary for concise, understandable specifications of events. A number of extensions can be suggested, but we defer discussion of this until chapter 7.

The event attributes have been selected for a number of reasons. Essentially, they allow the designer to access and manipulate the dialogue state. Some (out and prompt) reflect entries in the system part of the state (OUT$ and IR$, respectively). The when attribute defines the conditions under which the event may be selected. Other constructs in eventISL allow the specifier to manipulate application entries in the dialogue state.

An event has the overall form

```
event <EventName> =
    <attribute-list>
```

where <attribute-list> is a list of the attributes for the event and <EventName> is one of the event names from the eventCSP specification concerned. EventISL is formally defined in Appendix 7.

This section introduces eventISL by re-specifying the decision table example. In order to relate it to the idea of event operations, we show the results of translating the events into me too, but details of how this is done are left until chapter 5. As before, we assume that the dialogue state is held in the global object "dlg". For the purposes of illustration, we no longer hold the decision table in the system part of the state (in the DB$ entry). Instead, we assume that it is in the

application-specific part in an entry labelled "dt". Initially we also assume the existence of an operation to extract that entry:

```
getdt: DlgState -> Tree
```

4.2.1 Basic attributes of events

The first requirement is to be able to specify the condition operation for an event. This is given by the when attribute which defines when the event may be selected. The "is-decision" and "not-decision" events, for example, use only this attribute:

```
event is-decision =
   when is-decision(getdt(dlg))

event not-decision =
   when not is-decision(getdt(dlg))
```

where the attribute is defined by a boolean-valued me too expression. This expression becomes the body of the condition operation, so that the condition operations for the events above are

```
is-decision-C(dlg) ≡ is-decision(getdt(dlg))
not-decision-C(dlg) ≡ not is-decision(getdt(dlg))
```

The when attribute does not contribute in any way to the action operation for the event. If, as here, the event has no attributes contributing to the action operation, that operation has no effect on the state. The action operations for these events are

```
is-decision-A(dlg) ≡ dlg
not-decision-A(dlg) ≡ dlg
```

If the when attribute is omitted the condition defaults to true and the event may therefore be selected at any time.

The "give-decision" event illustrates a second attribute: the out attribute for returning output to the user. It is specified as

```
event give-decision =
   out getdt(dlg)
```

This attribute is defined by a me too expression yielding a value suitable for

output to the user. The expression sets the OUT$ entry in the state. Thus the operations for this event are

```
give-decision-C(dlg) ≡ true
give-decision-A(dlg) ≡ dlg ⊕ { OUT$→getdt(dlg) }
```

The last basic attribute for an event sets the IR$ flag in the state when input is required. This is signalled by the prompt attribute. Often it will be used in association with the out attribute (although this is not essential) as in the "prompt" event.

```
event prompt =
   out "?"
   prompt true
```

The event operations for prompt are

```
prompt-C(dlg) ≡ true
prompt-A(dlg) ≡ dlg ⊕ {IR$→true, OUT$→"?" }
```

These three attributes (when, out and prompt) are the basic attributes of a dialogue event. In the examples above, the values for the attributes have been constants or the result of some me too operation, but, as their syntax shows, their values can be determined by any valid me too expression yielding a value of the appropriate type.

4.2.2 Saving and retrieving objects

As in ECS, it is useful to be able to hold application-specific objects in the state. This section gives the eventISL constructs which allow these objects to be created and accessed by the application.

In the system-driven decision table example, for instance, it is appropriate to remember which question has been asked, so that the event involved is specified as

```
event ask-question =
   out question(getdt(dlg))
   prompt true
   qu = question(getdt(dlg))
```

This adds an entry named "qu" to the dialogue state which is used to save the value of "question(getdt(dlg))". The syntax of this expression is

```
<entry-index> = <Expr>
```

The action operation for this event is then

```
ask-question-A(dlg) ≡
        dlg ⊕ { IR$→true,
                "qu"→question(getdt(dlg)),
                OUT$→question(getdt(dlg)) }
```

Given a mechanism for saving objects in the state, we also need to be able to retrieve them. For this, eventISL provides the use expression which lists the entries required. Each entry index in the list becomes part of a me too let expression extracting its entry from the state. For example,

```
use X in <AttrExpr>
```

becomes

```
let X = dlg["X"] in <AttrExp>
```

The value of "qu" can be retrieved by

```
event user-answer =
use qu in
   dt = prune(getdt(dlg), qu, getinp(dlg))
```

Here we can also create and save a new value of the decision table, as can be seen from the action operation for the event.

```
user-answer-A(dlg) ≡
  let qu = dlg["qu"]
  in
      dlg ⊕ { "dt"→prune(getdt(dlg),qu,getinp(dlg)) }
```

Note that with use, we can now rewrite some of the event descriptions to extract the decision table directly:

```
event is-decision =
use dt in
   when is-decision(dt)

event not-decision =
use dt in
   when not is-decision(dt)

event give-decision =
use dt in
   out dt
```

Recall that the system entry IN$ holds user input. Events will require access to this entry, and so an event may refer to it by the index "input", as in

```
event user-answer =
use dt, qu, input in
    dt = prune(dt, qu, input)
```

The request for the "input" entry in the state extracts the IN$ system entry, so the action operation for this event is:

```
user-answer-A(dlg) ≡
    let dt = dlg["dt"]
        qu = dlg["qu"]
        input = dlg[IN$]<>
    in
        dlg ⊕ { "dt"→prune(dt,qu,input) }
```

The same idea applies to the DB$ entry which can be referred to as "db" within the specification.

4.2.3 Local declarations

If we rewrite the "ask-question" event as well, we obtain

```
event ask-question =
use dt in
   out question(dt)
   prompt true
   qu = question(dt)
```

This still requires "question(dt)" to be evaluated twice, so we extend eventISL by borrowing the let expression from me too to permit local declaration. We also add the retain expression, which saves a variable in the state. The event specification is now

```
event ask-question =
use dt in
   let qu = question(dt)
   in
        out qu
        prompt true
        retain qu
```

and has the action operation

```
ask-question-A(dlg) ≡
        let dt = dlg["dt"]
        in  let qu = question(dt)
            in
                dlg ⊕ { IR$→true, "qu"→qu, OUT$→qu }
```

Note that retain is simply an alternative way of expressing

```
"qu" = qu
```

4.2.4 Removing objects

The last construct in eventISL allows for the removal of objects from the state. It could be used with the "user-reply" event to ensure that user input is not held in the state after its use:

```
event user-reply =
use dt, input in
    dt = prune(dt, get-q(input), get-a(input))
    remove input
```

for which the action operation is

```
user-reply-A(dlg) ≡
    let dt = dlg["dt"]
        input = dlg[IN$]<>
    in
        dlg ds {IN$} ⊕
        {"dt"→prune(dt,get-q(input),get-a(input))}
```

Note that here, as in all event specifications, the event attributes and expressions can be written in any order.

4.2.5 Process initialisation

The next issue concerns the initialisation of the state (cf. the "start" operation in ECS). When the specification is executed, the interpreter is given all the necessary application-defined objects (ie. other than the dialogue objects shown in Fig.4.2) in a single argument. It saves this composite object in the DB$ entry in the state, referred to by the index "db". This is unlikely to be the most convenient form for the specification however, so a way is provided to initialise the state as required, namely by using a process specification. The process may specify no action, as in

```
process test
```

or it may perform some state-transformation and/or output. All the eventISL constructs except when are available and are translated into me too in the same way as event specifications.

For the decision table example, the process specifications might be

```
process dts =
use db in dt = db
```

```
process dtu =
use db in dt = db
```

to set up the table in the required entry in the state. Alternatively, the events could have referred directly to the "db" entry.

4.2.6 Process labelling in eventISL

Recall that in eventCSP a process P may be labelled by l by specifying it as l:P with labelled events l.e. In order to be used with eventCSP specifications containing labelled processes, eventISL must be able to specify labelled events. The syntax for this is

```
event <label>.<EventName> ... etc
```

as in

```
event ffe.prompt =
   out "?"
   prompt true
```

and

```
event dbfe.prompt =
   out "next command:"
   prompt true
```

4.3 SCHOLAR example

A more substantial example, which demonstrates the use of the parallel operator, is a specification of the SCHOLAR computer-aided instruction system [Carbonnell 70]. The original system exhibited a number of distinctive features, one of which is the style of interaction, where either partner (student or system) can take the initiative and ask questions of the other. In the example here, we are concerned to specify this style of interaction, rather than all the characteristics of SCHOLAR (such as use of natural language or inference from the data representation).

In a session with SCHOLAR, the student is asked questions and gives answers in much the same way as for a "drill-and-practice" CAI system. However at any point the student can, instead of answering the question, ask SCHOLAR for information. SCHOLAR responds as appropriate and then repeats the unanswered

question. Carbonnell coined the phrase "mixed initiative" to describe this style of interaction; in SCHOLAR, it can be characterised as a system-controlled style which allows interruption from the user. SCHOLAR also provides a user-controlled mode which can be requested by the student. In this mode, SCHOLAR answers questions posed by the student. With user input underlined, the structure of a SCHOLAR session might look like

```
Uruguay is a ? country
RIGHT
Peru has main language ? French
WRONG
Brazil is in ? QU Peru has main language
Spanish
Brazil is in ? QA
Confirm (y/n) ? y
? QU Bolivia
Bolivia is a country; Bolivia is in South America
? MI
Confirm (y/n) ? y
Brazil is in ? South America
RIGHT ...
```

First we give the eventCSP specification of SCHOLAR. This specification splits SCHOLAR into three subsystems. The first (MI) handles the mixed-initiative mode; the second (QA) handles the question-answer mode; and the third (switch) controls the switching between these two modes of interaction. These are specified as running in parallel with each other.

```
scholar = switch || MI || QA
   synchronised on { select-QA?, select-MI?,
                     select-QA, select-MI, initial-MI }

switch  = ( initial-MI → switch' )
switch' = ( select-QA? → confirm? →
                ( yes → select-QA → switch'
                [] no → select-MI → switch' )
          [] select-MI? → confirm? →
                ( yes → select-MI → switch'
                [] no → select-QA → switch' )
          )

MI  = ( initial-MI → mi
      [] select-MI → mi' )
mi  = (choose-question → mi')
mi' = (ask-question →
          ( select-QA? → MI
          [] user-answer → check-question → mi
          [] user-query → answer → mi' ) )
```

```
QA=  (select-QA → qa)
qa =  (prompt →
          ( select-MI? → QA
          [] user-query → answer → qa ) )
```

The SCHOLAR example illustrates the value of the parallel operator, since each subsystem can be specified separately. Indeed, any other interaction styles could be added relatively easily, involving only the switch process in any changes. Using || makes for a succinct, modular description of the system.

The eventISL specifications for SCHOLAR eventCSP are given below. As usual, the specification makes use of application types and operations. These are not given here but are specified in Appendix 8.

The AppState is defined by:

```
AppIndex = { "db", "qu" }
AppContents = ScholarDb U SchQu
```

with "db" holding the SCHOLAR information base and "qu" holding the current question. The process initialisation operation is

```
process scholar
```

which makes no change to the state, since all initialisation in this example is performed by the initial-MI event.

- events for the switch process

```
event initial-MI =
use db in
    db = initdb(db)
    qu = NullQu
```

The events which signal selection of a mode have no attributes, that is, they are synchronisation events and their use is controlled entirely by the structure of the eventCSP. They have no effect on the state.

```
event select-MI

event select-QA
```

The remaining events for the switch process all examine user input. For these events we assume the existence of a simple pattern-matching me too operation:

```
matches : Text x Pattern -> Boolean
```

(it is specified in Appendix 4 since it is also available in streamCSP).

Only two of the four event specifications are given here, due to their similarity:

```
event select-MI? =
use input in
   when matches(input,"MI")

event yes =
use input in
   when matches(input,"y")
```

- events for MI processes

```
event choose-question =
use db, qu in
     qu = if qu = NullQu then pickq(db) else qu

event ask-question =
use qu in
     out qu
     prompt true

event user-answer =
use input in
     when not ( matches(input,"QA")
                or matches(input,"MI")
                or is-question(input) )

event check-question =
use db, qu, input in
   out check(db,qu,input)
   db = register(db,qu)
   qu = NullQu
```

- events for QA and MI processes

```
event user-query =
use input in
   when is-question(input)

event answer =
use db, input in
   out query(db,input)
```

The prompt event from the decision table example (see §4.2.1) is reused for this specification, so we omit it here.

This completes the SPI specification of the SCHOLAR dialogue. It should be pointed out that this is the product of several iterations in the design, arrived at as the result of experimenting with the dialogue to ensure that it possessed the required features of SCHOLAR. From this example, we can see that the events for

a fairly sophisticated system like SCHOLAR can be specified in a straightforward way and that decomposing a system into its primitive events provides an effective modularisation of the system.

4.4 Form-based interaction

Using form-filling as a means of communicating with an interactive system has been explored by a number of researchers, eg. [Balbin et al 85] [Frohlich et al 85] [Hayes 85]. Advantages claimed for this approach include its flexibility, ease of construction and ability to offer a consistent interface across different applications. For our final example, we give a formal specification of part of such a system. [Studer 84] gives a high-level VDM specification of a forms-based system, but it is very abstract and not executable.

Here, a form is a sequence of single field entries, each of which can solicit one input from the user.

```
FormDb = ff(FormName,Form)
Form = seq(Field)
Field = tuple(FieldName,FieldAttr)
```

The definitions of other objects, such as the precise form of the field attributes, are not relevant to the dialogue specification and are omitted here. Suffice to say that the form structure allows the designer to specify default values, help texts, mandatory fields and inter-field dependencies. These are among a number of facilities recommended in [Gehani 83]. Appendix 9 specifies all the objects and operations in the forms component.

The AppState is:

entry index	type	used for
thisf	FormName	name of current form
lastf	FormName	name of previous form
f	Form	current form
flds	seq(Field)	fields to process
fld	Field	current field
done	seq(Field)	fields processed
fdb	FormDb	form database

The system first offers the user a choice of forms to fill:

```
forms = ( menu →
            ( valid-form → get-form → fill-in ; forms
            [] repeat? → get-form → fill-in ; forms
            [] inv-form → error → forms )
```

Only a selection of the events in this process are specified; the full specification is given in Appendix 10.

```
event menu =
use fdb in
    out form-menu(fdb)
    prompt true

event valid-form =
use fdb, input in
    when not matches(input,"REPEAT")
        and form-exists(fdb,input)
    thisf = input
    f = clear(get-form(fdb,input))

event repeat? =
use fdb,lastf, input in
    when matches(input,"REPEAT") and form-exists(fdb,lastf)
    thisf = lastf
    f = get-form(fdb,lastf)

event get-form =
use fdb, f, thisf in
    out display-form(fdb,thisf)
    done = <>
    flds = fields(f)
```

The fill-in process requests user input, allowing the user to supply a value, ask for help, skip the field, finish with the current form (with or without saving it) and to undo the previous field supplied:

```
fill-in = ( fields-left? → position → old-value
          → position → get-input →
                ( help? → fill-in
                [] skip? → fill-in
                [] cancel? → forms
                [] undo? → fill-in
                [] save? → check-form
                [] value? → update → fill-in )
          [] not-fields-left? → check-form )
```

Again, a number of event operations are omitted for the sake of space:

```
event fields-left? =
use flds in
    when not flds = <>
    fld = head(flds)
```

```
event skip? =
use flds, done, fld, input in
   when matches(input,"SKIP")
   flds = tail(flds)
   done = <fld> ^ done

event update =
use f, flds, done, fld, input in
   f = enter(f, get-name(fld), input)
   done = <fld> ^ done
   flds = remove-field(flds,fld)
```

Finally the check-form process ascertains whether or not all the required fields have been supplied.

```
check-form = ( complete? → save-form → skip
             [] not-complete? → fill-in )

event not-complete? =
use f in
   let to-do = not-complete(f)
   in
      when not to-do = <>
      out "error: some required fields not given"
      flds = to-do
      done = <>
```

The result of this event is that the user will be prompted for each remaining required field. The specification of this event illustrates how the design of the application and the dialogue cannot always be independent of each other, since the application originally returned just a boolean indicator as to the completeness of the form. After running the original prototype, it was decided to re-prompt for the missing fields which meant that extra facilities were needed in the application.

4.5 Summary

By drawing on the notation of CSP, the Read-Eval-Print paradigm and the concepts of finite-state machines, we have developed a two-layer model for specifying human-computer interaction, as in Fig.4.3.

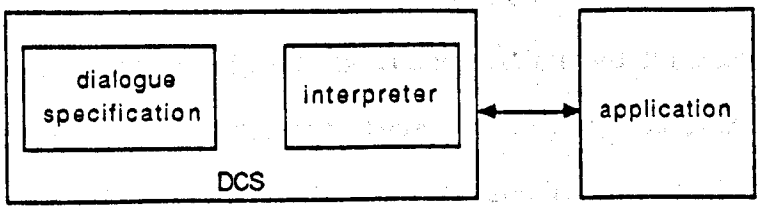


Fig.4.3 Dialogue control system - overview

EventCSP is based on CSP, using the general model of events without interprocess communication, and is intended as a language for specifying the overall structure of a dialogue. The activities which take place in that dialogue are specified by event operations written in a structured form of me too, namely in eventISL. Together they enable a dialogue to be specified and prototyped separately from the application; see Fig.4.4.

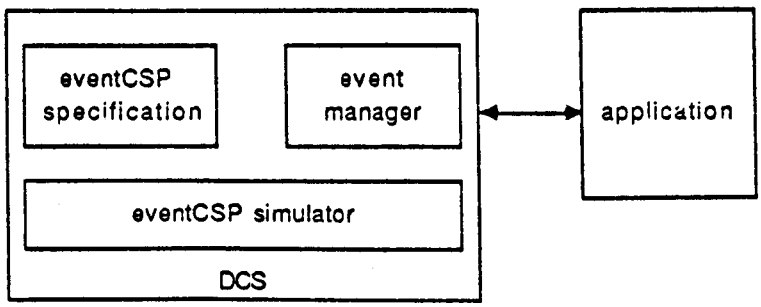


Fig.4.4 Dialogue control system – two layers

SPI fits into the UIMS model described in chapter 2 as a dialogue control system – the component which controls the interactions with the user on the basis of a supplied dialogue specification.

As a method, SPI forms part of the me too framework. Identifying the model, ie. the objects which are involved in user tasks and the operations upon them, remains as the first step in the method. These objects and operations are specified in me too, as before.

The dialogue designer, too, may well need to employ these steps in specifying objects and operations peculiar to the dialogue components, such as interaction histories or menu structures. In addition, though, the dialogue designer has to specify the structure and content of the dialogues to be offered to the user by the system. In specifying the dialogue events, the model of the application provides the task objects and operations available to the user through the interface.

With the specifications of both application and dialogue available, the entire system can be executed as a prototype. As before, this exercise is likely to suggest changes and reveal errors, so the method remains an iterative one.

SPI is thus seen as an extension to me too, not a replacement of it. It retains the me too iterative method; SPI's languages simply provide ways of imposing constraints on the structure and time of execution of me too operations.

As has been said already, the me too method and notation has been used to specify and prototype SPI. In the next chapter we give this me too specification of SPI.

CHAPTER 5

EXECUTING DIALOGUE SPECIFICATIONS

Given a SPI dialogue specification, consisting of an eventCSP description together with eventISL operations, the next task is to exercise the specification as a prototype. This chapter describes the tool which makes this possible - the SPI dialogue executor.

5.1 Overview of the dialogue executor

There are two central components responsible for executing the specification: the eventCSP simulator and the event manager.

The eventCSP simulator is an interpreter for the eventCSP portion of the specification. Its first function is to determine which events are currently possible according to the current position in the eventCSP structure. Secondly, given an event from this set of possible events, it uses that to move to the next position in the structure. The simulator can be invoked by the user, who is then responsible for selecting the event that is to happen. Alternatively, it can be called from the event manager which uses the condition operations for the events to select which events happen.

The event manager uses the eventCSP simulator to control which events may be triggered at any particular point in the dialogue. Given a set of possible events, it calls their condition operations and arbitrarily chooses one of the events for which this evaluates to true. Before moving to the next interaction point, it invokes the action operation for the chosen event to effect the required state transformation.

There are other components involved as well. For eventISL to be executed, the events it specifies have to be translated into the corresponding me too operations. This is done by a separate translation component when an eventISL specification is read. It is translated into the corresponding me too operations which are then defined as part of the run-time environment. No translation of

eventCSP is required since the specification itself acts as the data structure which is to be interpreted. Overall, the dialogue is controlled by a component which calls the event manager to execute the dialogue and interacts with the user as dictated by the dialogue state.

The relationships between these components are shown in Fig.5.1.

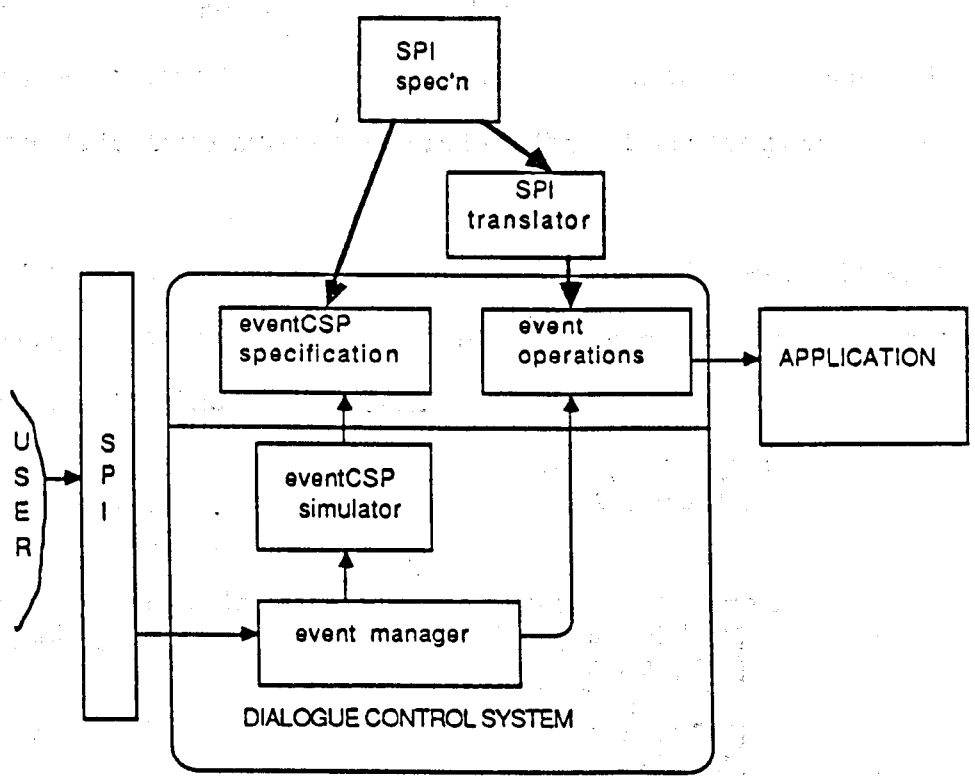


Fig.5.1 SPI dialogue executor

5.2 EventISL translator

A dialogue is specified in two parts - eventCSP and eventISL. The specification may be read from a file or it may be defined (or modified) interactively. The eventCSP part is incorporated into the SPI environment as a named data structure. The eventISL part is translated into its equivalent me too operations which are then added to the environment. This section describes how me too operations are created from the eventISL specification.

In order to create a me too operation, the translator constructs the text of the operation from the event description and then evaluates it to add the definition to the environment. Each event generates a condition operation and an action

operation, both taking the dialogue state as the sole argument. Thus an event "ex" would generate operations with functionality

```
ex-C : DlgState -> Boolean
ex-A : DlgState -> DlgState
```

Each construct in eventISL is translated into part of a me too operation. This section describes that translation for each construct, using the following notation:

if e represents some eventISL text, then
C[e]
represents its translation as required for the condition operation, and
A[e]
represents its translation as required for the action operation.

The translation rules are based on the formal definition of eventISL in Appendix 7, but the description below is illustrative rather than completely formal. Appendix 11 gives the formal definition of the rules.

```
C[event ex = E]          =    ex-C(dlg) = C[E]
A[event ex = E]          =    ex-A(dlg) = A[E]
where E is a DlgExpr
```

use is translated in the same way for both types of operation:

```
C[use a,b in E]          =    let a = dlg["a"]
                                  b = dlg["b"]
                              in C[E]
A[use a,b in E]          =    let a = dlg["a"]
                                  b = dlg["b"]
                              in A[E]
```

where E is an AttrExpr

The translation of let only changes the embedded AttrExpr E:

```
C[let x = e in E]        =    let x = e in C[E]
A[let x = e in E]        =    let x = e in A[E]
```

The special names for entries in the system part of the state are treated in the same way for both types of operation:

```
C[input] = A[input]      =    IN$
C[db] = A[db]            =    DB$
```

The remaining rules are based on the definition of an attribute list <AttrList> as

<AttrList> ::= empty | <AttrList> <Attr>

In the rules that follow, let E stand for an AttrList.

```
C[empty]                 =    true
A[empty]                 =    dlg
```

The when attribute is only used in condition operations. Its boolean-valued me too expression is unchanged by the translation.

```
C[E when B]              =    C[E] and B
A[E when B]              =    A[E]
```

C[E prompt B] = C[E]
A[E prompt B] = A[E] ⊕ { IR$→B }

C[E out text] = C[E]
A[E out text] = A[E] ⊕ { OUT$→text }

C[E retain x,y] = C[E]
A[E retain x,y] = A[E] ⊕ { "x"→x,"y"→y }

C[E x = e] = C[E]
A[E x = e] = A[E] ⊕ { "x"→e }
where e is an Expr

Removing entries from the state is achieved by subtracting the entry indices from the domain of the state.
C[E remove u,v] = C[E]
A[E remove u,v] = A[E] ds {"u", "v"}

5.3 EventCSP simulator

EventCSP is implemented by an interpreted data structure. This structure is a process database which is a finite function mapping process names to their definitions. Simulating the behaviour pattern of a process is achieved in two steps. For the current process definition, the set of possible next events is determined. Of these, one is selected by some means (to be discussed later) and is used to advance one step in the definition.

First we consider how to determine the possible events by specifying the behaviour of a "nextevents" operation for each construct in the language.

nextevents : Process x ProcessDb -> set(EventName)

For brevity, the me too definition of nextevents is given using pattern-matching on the language constructs to distinguish the cases and to name the constituent parts of each construct:

nextevents((a→P), pdb) ≡ {a}

nextevents(P [] Q, pdb) ≡
    nextevents(P,pdb) U nextevents(Q,pdb)

nextevents(Pname, pdb) ≡ nextevents(pdb["Pname"],pdb)

nextevents(skip, pdb) ≡ { TICK }

nextevents(P;Q, pdb) ≡
    if TICK ∈ nextevents(P,pdb)
    then nextevents(Q,pdb)
    else nextevents(P,pdb)

```
nextevents( l:P,  pdb) ≡
      { (l.e) | e ← nextevents(P,pdb) }

nextevents( P||Q,  pdb) ≡
      let N1 = nextevents(P,pdb)
          N2 = nextevents(Q,pdb)
          S = synchronisers(P||Q)
      in
          (N1-S) U (N2-S) U (N1 ∩ N2 ∩ S)
```

where "synchronisers" returns the set of events on which P and Q are synchronised.

Given a set of possible events, one is chosen and used to advance one step in the current process definition. (How that choice is made depends on whether the event manager or the user is driving the simulator, an issue that will be addressed in later sections of this chapter.) The simulator uses an operation called "step" to move on, given the chosen event:

```
step : Process x EventName x ProcessDb -> Process
```

As before, we use me too with pattern-matching for brevity:

```
step( (a→P), ev, pdb) ≡ P

step( P [] Q, ev, pdb) ≡
          if ev ∈ nextevents(P,pdb)
          then if ev ∈ nextevents(Q,pdb)
               then step(P,ev,pdb) [] step(Q,ev,pdb)
               else step(P,ev,pdb)
          else step(Q,ev,pdb)
```

(Note that this implements a "benevolent" non-determinism which does not choose between alternative processes until forced to do so.)

```
step( Pname, ev, pdb) ≡ step(pdb["Pname"],ev,pdb)

step( skip, ev, pdb) ≡ abort

step( P;Q, ev, pdb) ≡
                if TICK ∈ nextevents(P,pdb)
                then step(Q,ev,pdb)
                else step(P,ev,pdb);Q

step( l:P, l.ev, pdb) ≡ l:step(P,ev,pdb)

step( P||Q, ev, pdb) ≡
      let N1 = nextevents(P,pdb)
          N2 = nextevents(Q,pdb)
          S = synchronisers(P||Q)
      in
          if ev ∈ N1 ∩ N2 ∩ S
          then step(P,ev,pdb) || step(Q,ev,pdb)
          else if ev ∈ N1
          then step(P,ev,pdb) || Q
          else P || step(Q,ev,pdb)
```

These two operations, nextevents and step, form the core of the eventCSP simulator. The simulator maintains a state to control its activities. It is defined as the me too object

SimState = tuple(ProcessDb,Process,set(EventName),Msg)

where the first element is the original eventCSP specification (the process database), the second is the current process definition, and the third is the set of possible next events for that definition and the last element is used to give messages (errors, prompts, or menus) to the user.

ProcessDb = ff(ProcessName, Process)
Process = concrete syntax of eventCSP (see Appendix 6.2)
Msg = seq(Atom)

The controlling operations for the simulation are

initstate : Process x ProcessDb -> SimState
nextstate : SimState x EventName -> SimState

and are specified as

initstate(p,pdb) ≡ (pdb,p,nextevents(p,pdb), "PICK ONE OF:")

```
nextstate(ss,e) ≡
   letrec (pdb,p,n,msg) = ss
   in
      if e ∈ n
      then let p' = step(p,e,pdb)
           in (pdb,p',nextevents(p',pdb), <"OK- PICK ONE OF:">)
      else (pdb,p,n,<"WRONG- PICK ONE OF:">)
```

An additional operation to show the appropriate part of the state to the user is:

showsim(ss) = fourth(ss) ^ sort(third(ss))

The simulator as described above is used unaltered by the event manager to prototype dialogues. For direct use, the user employs a number of special commands which create and manipulate the simulator state and display it on the screen, as illustrated in chapter 3.

5.4 Event manager

The eventCSP specification can be run as a prototype, not just a simulation, by linking it with the event operations specified in eventISL.

To do this, the event manager uses a state which amalgamates the simulator state and the dialogue state (see §4.1.1). The entire simulator state is included as a single entry in the dialogue state to ensure that the simulator can be run unchanged. It is extracted from the overall dialogue state by

simstate(dlg) = dlg[SS$]

The operations in the event manager correspond to those in the ECS cycle. Fig.5.2 shows the corresponding operations.

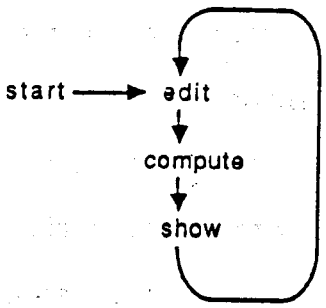


Fig.5.2(a) ECS execution cycle

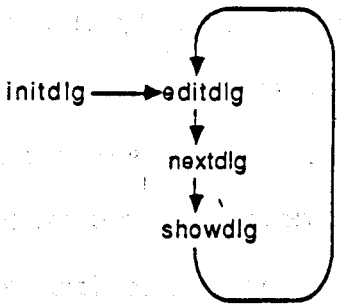


Fig.5.2(b) SPI execution cycle

The major difference is that the dialogue designer does not have to supply any operations in the cycle explicitly, since these are all part of the event manager. The cycle is implemented by the SPI interpreter; the event manager simply supplies the operations needed for it. In this section, we specify the major operations of the interpreter, as shown in Fig.5.2(b). Subsidiary operations are specified in Appendix 13.

Initialisation is accomplished by the initdlg operation which has functionality

initdlg : ProcessName x ProcessDb x InitAppState -> DlgState

and which sets about the initialisation in two steps. First it creates a new dialogue state containing the application state given and a new simulator state (created by the call on the simulator initstate operation). Secondly, it invokes the process initialisation operation for the named process. The "call-action" operation constructs the name of the action operation from a process or event name and calls it to perform the state transformation. The operation is specified by

```
initdlg(proc-nm,pdb,udb) =
    let dlg = { SS$→initstate(pdb[proc-nm]0,pdb),
                DB$→udb }
    in call-action(proc-nm,dlg)
```

User input is added to the state by:

```
editdlg(dlg,text) = dlg ds {IR$} ⊕ { IN$→text }
```

and output is shown by

```
showdlg(dlg) = dlg[OUT$]"no output"
```

The bulk of the work is performed by the "nextdlg" operation. This uses the simulator state and the event operations to control the dialogue. If the simulator indicates no further progress is possible, the termination flag is set and no more is done. Otherwise the condition operation for each of the possible events is evaluated against the state and one chosen arbitrarily. For the chosen event, its action operation is invoked, a new simulator state is created and the new version of the dialogue state is returned. If no event is possible, a special error event is returned. This event has a system-defined action operation which is called to give an error message to the user.

```
nextdlg(dlg) =
    letrec ss = simstate(dlg)
           (pdb,p,n,msg) = ss
    in
        if process-end(p,n)
        then dlg ⊕ { STOP$→true }
        else let e = choose-event(n,dlg)
             in call-action(e,dlg) ⊕ { SS$→nextstate(ss,e) }
```

As with the eventCSP simulator, the event manager can be run directly by the user with various commands but in practice it is invoked by the SPI interpreter.

5.5 The SPI interpreter

As in ECS, the interpreter controls the dialogue by calling the underlying operations and interpreting the dialogue state. After initialisation, a loop is entered to repeatedly accept any input, create a new version of the state and show any output generated as a result. This continues until the termination flag is set.

This component is currently written in Lisp and used to replace the

Read-Eval-Print mechanism in the Lisp system. However, here we outline its specification in SPI (omitting the less important event specifications for the sake of space):

```
rep = (header → rep1)
```

The rep1 process requests the parameters for the call to "initdlg" which is made in the init event.

```
rep1 = (get-params → init → rep2)
```

The rep2 process detects when the termination flag is set and offers a choice between rerunning the shell or finishing.

```
rep2 = ( dlg-end? → end-run → options →
                        ( restart? → rep1
                        [] finish? → exit → skip )
       [] not-dlg-end? →
                        ( input? → in → rep3
                        [] not-input? → rep3 ) )
```

At this meta-level of description, we cannot fully specify events in eventISL since the events for the interpreter deal with the system entries in the dialogue state and with the state as a whole. To describe the behaviour of the interpreter, therefore, we have to allow ourselves the licence to use and set the dialogue state "dlg" in the events.

```
event input? =
    when dlg [IR$] false
    prompt true

event in =
use input in
    dlg = nextdlg(editdlg(dlg,input)) ⊕ {IR$→false}

event not-input? =
    when not dlg[IR$]false
    dlg = nextdlg(dlg)
```

The rep3 process is responsible for giving output from the state if any is present.

```
rep3 = ( output? → out → rep2
       [] not-output? → rep2 )

event out =
    out dlg[OUT$]<>
    dlg = dlg ds {OUT$}
```

This interpreter is adequate for prototyping purposes, although clearly there are many improvements that could be made (such as to allow direct, menu-driven interaction with the eventCSP simulator or single-stepping the event manager for debugging purposes, and so on). At this stage, such enhancements are unnecessary, but they will have to be considered for a full production version of this tool.

One extension has been made to the system, however. This is based on the concept of traces found in CSP [Hoare 85] and provides a way of monitoring the events occurring in a dialogue as it is executed. This development is described in the next section.

5.6 Traces

> "A trace of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment of time."
>
> [Hoare 85, p.41]

For example, for the process

$$P = (a \rightarrow (b \rightarrow P \,[]\, c \rightarrow P))$$

the traces of P include

$$\langle a\ b\ a\ b\ a \rangle \quad \text{and} \quad \langle a\ c\ a\ b\ a\ c \rangle$$

but not $\langle a\ a \rangle$ or $\langle a\ b\ c \rangle$

Hoare uses traces to characterise processes for much of the mathematical theory of CSP. In SPI, they can be used more practically as a means of monitoring the progress of a dialogue.

Since part of the rationale for prototyping interaction is to allow the prototype to be subject to some form of trials and experiments with other designers and potential users, clearly evaluation of those experiments is necessary to provide the feedback for the next iteration of the design. The exact nature of that evaluation is the subject of much discussion in the human factors literature (see, for example, [Bleser & Foley 82] [Good et al 84] [Lindquist 85]). Nevertheless, one component of evaluation is widely accepted: the dialogue prototype should provide

some means of recording the activity of its users [Williges 84], perhaps for analysis or "re-playing" the dialogue [Neal & Simons 83].

Traces in CSP offer a simple model for logging each event as it happens. In a practical tool, of course, this would involve adding information to each entry in the log, such as timestamps, but for now we simply extend SPI to record each event.

This extension requires no changes to the eventCSP or eventISL languages, but to the eventCSP simulator and its state. The simulator state is extended to include two new entries: a boolean flag indicating whether or not tracing is required and the trace itself, which is defined as

```
Trace = seq(EventName)
```

so that SimState is now

```
tuple(processDb, Process, set(EventName), Msg, Boolean, Trace)
```

The trace is maintained by the nextstate operation (§5.2.2) which is now specified as:

```
nextstate(ss,e) =
    let (pdb,p,n,msg,trf,tr) = ss
    in
        if e ∈ n
        then let p' = step(p,e,pdb)
                 tr' = if trf then tr^<e> else tr
             in (pdb,p',nextevents(p',pdb), "OK- PICK ONE OF:",trf,tr')
        else (pdb,p,n,"WRONG- PICK ONE OF:",trf,tr)
```

As far as the user interface to SPI is concerned, we introduce two meta-commands to the SPI shell, one to switch tracing on and off and another to extract the trace from the state.

We could also implement some of the CSP trace operations to provide a trace analysis package:

tr⌈A	restricts trace tr to symbols in the set A
s in tr	determines whether or not s is a subsequence of tr
#tr	yields the length of tr so that #(tr⌈A) gives the number of occurrences in tr of symbols in A
s⌈x	= #(s⌈{x}) to count the occurrences of symbol x

Together with the usual me too operations on sequences, these operations provide some tools for analysing the events in a monitored dialogue. It would also be possible to allow "replays" of a dialogue using trace files.

5.7 Summary

The combination of SPI and me too described so far allows a software designer to specify many aspects of an interactive system: its functionality, the structure of its interactions with users, and input and output formats. The current tools are adequate for the stated purpose, specifying and prototyping dialogues, but many additions and improvements are immediately apparent.

The me too/Lisp-based implementation of SPI is, however, only a prototype, limited in its functionality and in its performance. In developing SPI, we have employed the me too method in an iterative process of designing a software product. In this case, the product happens to be a tool for specification and prototyping. Having specified and prototyped its design, and redesigned it in the light of experience with the prototype, we are now in a position to implement a production version. The next chapter discusses such an implementation.

CHAPTER 6

TOWARDS A CONVENTIONAL IMPLEMENTATION

Thus far, we have sought to demonstrate the viability of the SPI languages for dialogue specification by applying them to a variety of examples. This chapter takes a different approach but with the same end in mind. In order to demonstrate that the SPI architecture and languages provide a sufficiently complete design for dialogue specification, we show how the system can be implemented by an imperative language on a conventional system. The implementation described here is not seen as the final version, but as the first steps in that direction.

The previous chapter showed how a SPI dialogue specification is executed, and described the various components of the dialogue executor. Some were specified in me too (and are executable as a result) while others were written in Lisp. Together, they act as a prototype of the dialogue executor. They make few concessions to usability and show barely adequate performance, features acceptable in a prototype of the tools but not in the tools themselves. Implementing the languages and the dialogue executor in C under UNIX enabled some of these issues to be addressed.

In this chapter we show one way in which SPI can be implemented as a set of usable tools with acceptable performance, and discuss some of the implementation decisions which were taken.

6.1 Initial implementation decisions

Some early decisions affected much of the way in which the implementation was tackled and so these are presented here before the individual components are described.

The first decision was to follow the me too specifications for the eventCSP simulator and event manager quite closely, retaining both the internal structure of the various operations and the overall architecture. Thus the simulator is implemented independently of the event manager and, as in the prototype, can be

run separately to simulate the dialogue. The event manager calls functions in the simulator, and is in turn called by the SPI interpreter.

In the same way that the computational structure of the prototype was retained, so also many of the data structures were kept. In particular, sets were considered a useful data type and so a module implementing sets (as linked lists) was provided.

The major difference in data structure concerned the representation of eventCSP. In the prototype, the eventCSP specification was represented by a finite function mapping process names to their definitions. In the implementation described here, a single process definition is created, with pointers to processes replacing process names, for reasons given in the next section. This change is responsible for the majority of the differences between the prototype and the implementation.

6.2 Processing the eventCSP language

One method of implementing eventCSP is to translate it into a more conventional language. This approach is taken in implementing "squeak", a language incorporating many of the features of CSP [Cardelli & Pike 85]. Amongst other things, this involves expanding the parallel construct to allow all the interleavings of events that this expresses.

Alternatively, eventCSP can be implemented as an interpreted data structure, as in the SPI prototype. This is not dissimilar to the extension to Hope proposed in [While 86], which employs a data structure to control the use of Hope recursion equations. Here we use the eventCSP data structure to control the invocation of me too operations or C functions. This is the method employed in the current version of SPI.

In the me too prototype, the S-expression version of the eventCSP specification itself acts as the data structure. In a conventional language such as C, the symbol manipulation involved in maintaining a textual version of the structure is

inefficient and so it was decided to create a pointer version of the structure instead. Where the prototype refers to called processes by name, the implementation uses pointers to process definitions. An eventCSP specification is "compiled" into this data structure, ready for use by the simulator. It can also be saved in text form in a file so that it can be kept after compilation.

Some changes to the notation were made to ease implementation. Firstly, by way of concession to the ASCII character set, the choice operator is denoted by ^, as in P ^ Q and arrows by ->, as in (a -> P).
Secondly, the process definitions should be fully bracketed. Thirdly, the synchronisation events for the parallel operator have to be given explicitly. These are the events common to both processes, so if we have

```
P = ( a -> b -> P )
Q = ( b -> c -> Q )
```

then the set of common events for P and Q is {b} and the parallel operator would be written thus:

```
( P || Q {b} )
```

Finally, the C restrictions on identifier names have to be noted: an identifier must begin with a letter but subsequent characters may be alphanumeric. Underscore (_) is regarded as a letter. In most implementations of C, the significant portion of an identifier is restricted to the first few characters (8 in C under PNX). Longer event names may be used but the designer should be aware that they will be truncated by the C compiler.

Allowing for these modifications, the eventCSP specifications given in this thesis have all been compiled and run on the simulator. For example, the eventCSP specification for decision tables becomes

```
dts = ( (is-decision -> (give-decision -> skip))
      ^
        (not-decision -> (ask-question -> (user-answer -> dts)))
      )
```

The syntax for the C version of eventISL is defined in Appendix 6.

EventCSP has been implemented using LEX and YACC [Johnson & Lesk 78]. As each eventCSP construct is recognised by the YACC-generated parser, it is treated as an internally-named process and entered in a temporary process database. When the entire specification has been read, this process database is either transformed into the required process definition or it is written to a text file for later use.

Translating an eventCSP specification to its internal form and then to a process definition is achieved with very acceptable performance.

6.3 EventCSP simulator

This section outlines how the me too specification of the simulator was used to guide its implementation. As in the prototype, the heart of the simulator is the pair of functions, nextevents and step. The computation and structure of these functions are the same as for their me too counterparts. Most differences arose from the representation of processes as a single process definition rather than as a database of named definitions. The implementation retains the benevolent non-determinism of the prototype.

The simulator maintains a state corresponding to the SimState of the prototype. It no longer needs to keep a copy of the process database, so this component of the state is omitted. The state is represented by global variables in the simulator module. The "current process definition" is actually a pointer into the dynamically-extended process definition.

The control loop of the simulator again reflects that of the prototype, offering the possible events, accepting a choice of event and stepping on to the next position in the process definition. Termination is defined as being when the single event "TICK" is offered, ie. when a skip process is encountered that is not part of an enclosing parallel or sequence process.

It can be seen from this outline of the simulator that the decision to follow the prototype so closely made its implementation a straightforward matter. However, although it was convenient and the result outperforms the prototype, it is not

space-efficient, building up extensions of the process definition during execution. This is due to the change in process representation.

There are, of course, alternative implementation strategies. This particular one was almost entirely determined by the early decisions described in §6.1. Different decisions at that point would have resulted in a different implementation. We regard this implementation as one of a number of possible "refinements" of the specification. It is not necessarily the best, but it has the merit of being constructed quickly. In its own way, the implementation is a prototype, the next step in the evolution from initial requirements ("a way of specifying and prototyping hci") to a fully-fledged set of tools capable of meeting those requirements.

6.4 Processing the eventISL language

In the prototype, eventISL is embedded in me too; that is, its constructs use me too expressions and the specification is translated into me too operations. In the implementation, eventISL is embedded in C, so that its constructs use C expressions and the specification is translated into a C program. Although the approach is the same, the differences between a functional specification language and an imperative programming language mean that there are differences between the me too and C versions of eventISL. This section is concerned simply with outlining the changes to the notation and the reasons for them.

The syntax is defined in Appendix 7 and, to illustrate some of the differences, the C version of the eventISL specification for the forms example is given in Appendix 10. In §5.2.1, we listed the rules for translation that are employed in the prototype; the corresponding rules for the implementation are given in Appendix 12.

Many of the changes result from the fact that values can be held in ordinary variables, using the normal assignment and access mechanisms provided in C. Consequently, using an explicit dialogue state to hold system and application objects is no longer necessary. The system part of the state is declared as variables within the event manager module. The application part is declared by the designer as C data within the eventISL specification.

Being able to store and access data in the state directly in C means that we no longer require explicit language constructs for this in eventISL. The use expression becomes redundant, and the saving of values is achieved by assignment to application variables. This implies the need to incorporate C code into the body of an event, since we now need to use C assignment statements. A new attribute, text, has been added to allow this; an event may have more than one text attribute. It should be noted that order is significant in an imperative language and so, unlike the me too version, the order in which attributes are given in an event becomes significant.

The use of text attributes in an event also removes the need for the let expression. The event

```
event user-reply =
use input, dt in
    let uq = get-q(input)
        ua = get-a(input)
    in
        dt = prune(dt,uq,ua)
```

can be written as

```
EVENT user_reply
TEXT uq = get_q(input);
       ua = get_a(input);
       dt = prune(dt,uq,ua);
```

in the concrete syntax of the C version of eventISL.

Using variables to represent the state means we no longer have the option to remove data from the state, since C does not allow variables to be "un-declared" once declared. Instead, a specification must set a null value in a "removed" variable to signal the non-availability of data.

As a result of these changes to the language, an eventISL specification now only needs to use the basic attributes (when, out and prompt) together with the new text attribute. Of these, the boolean-valued attributes are used in the same way as before, being supplied now with a boolean-valued C expression.

A number of options were possible for the out attribute, since C provides a number of ways of constructing text output. The choice made was that the

attribute should supply the text in the form required by the "printf" function, giving a format and data to be output. This is an experimental decision, and open to change as the language is used. In particular, we will require some mechanism to allow the output of graphical as well as textual information.

A minor change, made necessary by the requirements of C, is that all application functions must be declared as external functions before use.

As an example, we give the C version of one of the decision table specifications in Fig.6.1.

```
DIALOGUE
/*****
 *
 *          dts - eventISL specification
 *
 *****/

#include "dt.h"

extern unsigned check_decision();
extern char *question();
extern DT_PTR prune();
extern DT_PTR dt_example();

/**
 *          declarations for decision tables
 **/
DT_PTR dt;
char qu[256];

PROCESS dts
          TEXT dt = dt_example();

EVENT is_decision
          WHEN check_decision(dt)

EVENT not_decision
          WHEN !check_decision(dt)

EVENT give_decision
          OUT "\ndecision: %s\n",dt->text

EVENT ask_question
          TEXT strcpy(qu,(char *)question(dt));
          PROMPT TRUE
          OUT "\n%s ? ",qu

EVENT user_answer
          TEXT dt = prune(dt,qu,input) ;
```

Fig.6.1. Decision table example - C version

An example of how the screen display would appear during execution of this dialogue is given in Fig.6.2

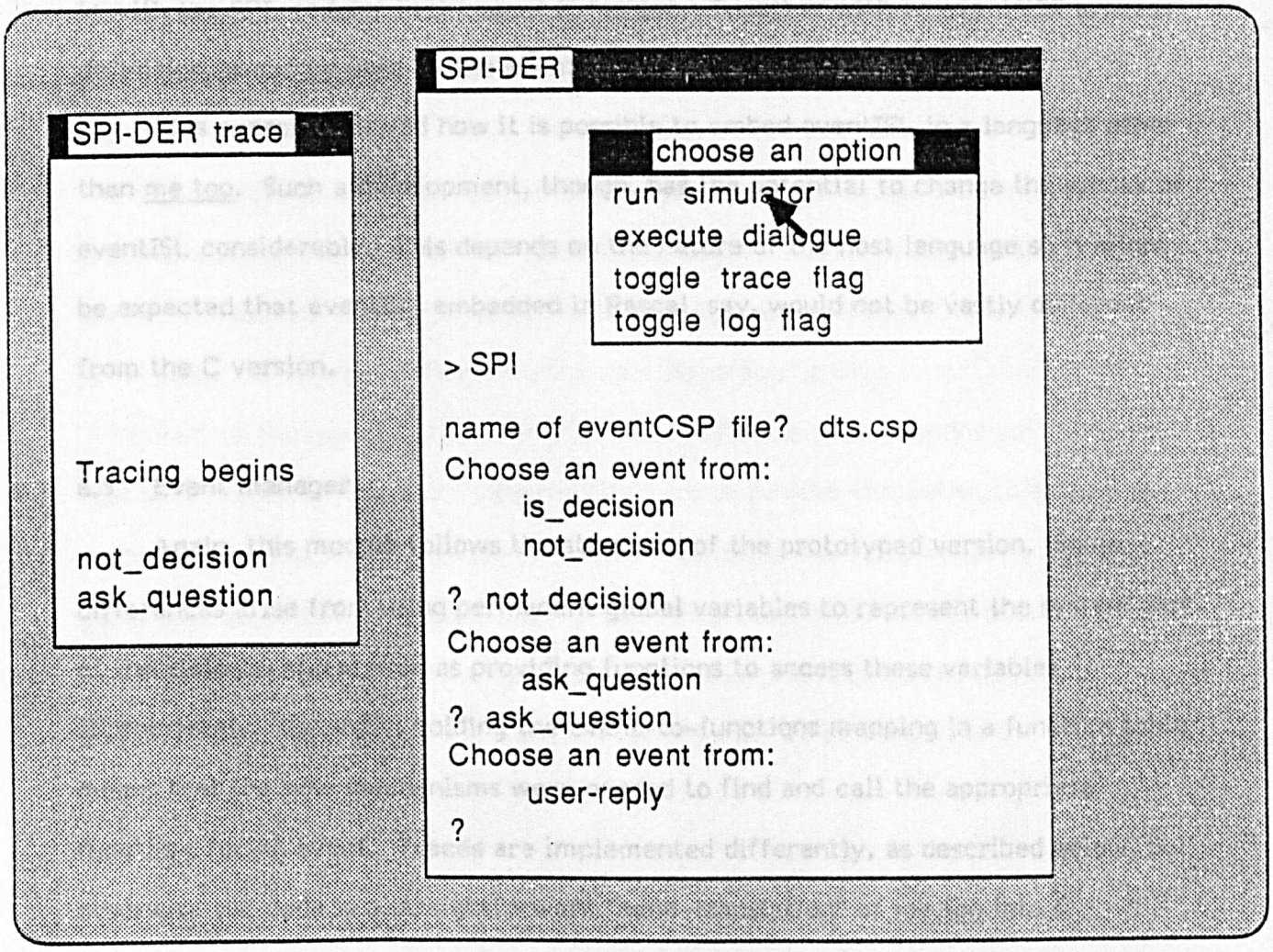


Fig.6.2 SPI screen display

The translation of eventISL is also implemented using LEX and YACC, and follows a similar pattern to that in the prototype. Each event causes the creation of two C functions – a condition function and an action function. Each attribute causes a fragment of C to be added to the generated program as part of the appropriate function. For ease of implementation, we impose the restriction that the when attribute, if present, must precede all other attributes for that event.

Some additional C code is also required. Various system files are "included" into the program and links to the system part of the state are established. At the

end of the program, the translator defines a function table which maps the event names to their condition and action functions.

The program generated by eventISL translation has to be compiled and linked in with the SPI modules in the usual C fashion. Together with an eventCSP process definition, it can be used by the event manager to prototype dialogues.

This exercise showed how it is possible to embed eventISL in a language other than me too. Such a development, though, has the potential to change the syntax of eventISL considerably. This depends on the nature of the host language so it might be expected that eventISL embedded in Pascal, say, would not be vastly different from the C version.

6.5 Event manager

Again, this module follows the structure of the prototyped version. Some differences arise from using permanent global variables to represent the system part of the dialogue state, such as providing functions to access these variables appropriately. Secondly, holding the event-to-functions mapping in a function table means that the new mechanisms were needed to find and call the appropriate functions for an event. Traces are implemented differently, as described below, but otherwise the code is a straightforward "hand-translation" of me too into C.

The trace of the execution of a process is no longer held as an entry in the state. In a practical tool, we recognise that there are (at least) two different uses for a trace of execution. First, for monitoring or feedback purposes, we require a time-stamped permanent log recording not only each event but, at a minimum, the user input supplied as well. On the other hand, for debugging, a dynamic display of each event selected is probably sufficient. Accordingly, we distinguish between logging events (to a file) and tracing events (to a screen window). The screen shown in Fig.6.2 includes the trace window.

Where, in the prototype, an event was added to the sequence held as the trace, now calls are made to a logging function and a tracing function which add the event

to the log file and trace window respectively. In addition, when an input is received, another logging function is called. Entries in the log and output of trace only occur when their respective flags have been set by user commands. These commands, as before, are processed by the SPI interpreter, which is described in the next section.

6.6 The SPI interpreter

In chapter 5, we gave an outline of how the dialogue state is interpreted in order to allow prototyping of the specified dialogue. Although this module followed the same structure (the Read-Eval-Print paradigm essentially), it also afforded some scope for experimenting with the user interface to SPI.

Part of the module implements the specified interpreter as one option which may be selected by the user. Other options are to run the simulator, to toggle the trace flag, to toggle the log flag or to quit. All options are presented in a pop-up menu with selection by mouse button press. There are other possibilities which could be implemented, such as dynamic creation or modification of dialogue specifications, but for now the front-end is adequate to demonstrate the tool.

6.7 Summary

This chapter shows how we have begun to address the implementation of SPI using an imperative language in a conventional system. A number of questions are raised by such a process, as is to be expected when moving from specification to implementation. The choice of representation for data structures, improving efficiency, maintaining a correspondence between specification and implementation, considering the user interface in more detail: these are all traditional concerns at this point in product development and all have been touched upon in the discussion above.

Implementing SPI in this way has, as required, improved both its performance and its presentation. These are subjective judgements, based on experience with both the prototyped and implemented versions. The difference is marked, even if not measured.

In one instance, namely traces, the implementation differed from the specification, as practical experience was gained with SPI. These differences were anticipated in the specification, which stated what was required (a tracing mechanism) and an approach (based on event names as selected) while acknowledging that more detail would be needed for the implementation.

As it stands, this implementation incorporates the functionality of the prototype. Experience with the system, though, has revealed that a number of enhancements are desirable. These include dynamic creation and modification of SPI specifications, the ability to single-step through a dialogue, the establishment of libraries of interaction techniques and tools for using and analysing log files. These are important features if SPI is to be of use in practical situations, but the present implementation suffices to show that the SPI architecture has tackled the major issues, that it can be implemented within a conventional system and that it has the potential for use in product development.

CHAPTER 7

COMPARISONS AND CONCLUSIONS

This chapter reviews the work reported in this thesis. We begin by comparing SPI with other techniques advocated for specifying and prototyping human-computer dialogues. Before concluding the thesis with a summary of what has been achieved with SPI, we suggest ways in which SPI could be used and extended.

7.1 Comparisons with other techniques

This section compares SPI with techniques advocated by other authors. We select some of the significant features found in these other methods and examine SPI in the light of them.

Many of the methods can be regarded as data-based, describing all (or a significant amount) of the dialogue in some data structure, perhaps augmented by actions or with separate control of the sequencing, as with the frames of [Lafuente & Gries 78]. This data structure may consist of such objects as BNF rules, state transition diagrams, frame descriptions or interaction event tables. It is then processed by an interpreter or is used to create an interpreter, thus providing a prototype of the dialogue described. SPI adopts a similar approach, but uses two separate data structures: the dialogue state and that derived from the eventCSP specification.

The first of these, the dialogue state is defined by the actions of the events and is processed by the SPI interpreter. Thus it is a dynamic description of the dialogue, subject to change in each cycle of the dialogue. In the other methods mentioned, the data structure is static, its contents defined at the outset. An advantage of the static approach is that the data structure describing the dialogue can be used to drive on-line help facilities automatically. One system [Feyock 77] uses the state transition diagrams to answer such questions as "what are valid commands in the current state?" or "how can state X be reached from here?".

In SPI, the second data structure is the representation of the eventCSP specification, which is static and could be used as the basis for similar analytic tools.

Several methods consider a dialogue to be made up of distinct steps, each step having various pre-determined characteristics. Examples are state transition diagrams, interaction events and DMS [Hartson et al 84]. SPI takes a similar view of the dialogue, considering each event as having the potential properties of input, action (state transformation) and output.

In a production system, rules are specified to determine the actions taken in the dialogue, depending on the contents of the working-memory. If these actions include modification of the rules themselves, then it is possible to develop dialogues which adapt to the behaviour of the user. The CONNECT state-transition-network system [Alty & Brooks 85] offers a measure of adaptability by extending the dialogue description to include production rules which can be used to modify the network. SPI does not yet address this issue of adaptability.

Another advantage claimed for production systems is that they make it possible to describe dialogues where there is no ordering or only a partial ordering on events. Other notations offering such capabilities include statecharts, flow expressions and the supervisory cells of SUPERMAN [Yunten & Hartson 85]. The parallel operator in eventCSP allows SPI specifications to give a similar degree of flexibility in dialogues. This is particularly useful in specifying concurrent input and direct manipulation screen-based interfaces.

The ability to specify interactions by composing, or bringing together, smaller specifications is evident in most of the methods. For some, such as notation based on CLG [Browne et al 86] and GUIDE [Gray & Kilgour 85], it is fundamental to the method since they are based on hierarchic structuring; in GUIDE, the hierarchy is based on the UNIX filing system. In other methods, like state transition networks, it is a feature which as been added to enable the specification to be decomposed into smaller, more comprehensible parts. For SPI, the unit of specification is the process. These units can be combined as defined in the eventCSP language.

SPI shares with EPROS [Hekmatpour & Ince 86a] and UML/GUSL [Green 85]

the goal of being able to specify all aspects of an interactive system within a single framework. SPI achieves this by adding notation for dialogue control to me too; the result is that both the application and the dialogue can be specified within this augmented me too method.

A number of techniques distinguish the two layers identified in SPI: the structure and the content of a dialogue. Examples include the systems described in [Christensen & Kreplin 84] and [Lafuente & Gries 78]. ADDS makes a similar division [Burns & Robinson 86], defining a dialogue using "scripts" and state transformations. EventCSP is a more powerful notation than the scripts they describe, however.

Finally, some methods, like state transition diagrams, are particularly appropriate for conveying the sequence of events in a dialogue. CSP was chosen as the basis for one of the SPI languages because it was found to share this property. Abstracting from the details of the dialogue and expressing the resulting abstract structure in eventCSP clearly shows the possible sequences of events in the dialogue. Moreover this structure can be explored interactively using the eventCSP simulator.

7.2 Suggestions for further work

Comparing SPI with these other methods suggests a number of ways in which this work might be extended.

7.2.1 Analysing dialogue specifications

Analysing formal specifications of dialogue is a useful technique for a number of reasons. It can be a way of determining whether or not the dialogue specifications meet various guidelines [Bleser & Foley 82] [Anderson 86], finding paths through the dialogue [Brown 82] [Alty 84], estimating performance times [Card et al 80] and predicting potential user reaction [Reisner 83] [Lindquist 85]. CSP is a formal language that also lends itself to analysis and derivation of properties of specifications [Goltz & Reisig 84] [Barringer et al 85] [Hoare 85]. It would be useful

to provide tools for analysing dialogue specifications written in eventCSP.

7.2.2 Extending event descriptions

As a language, eventISL contains only as much as is necessary for dialogue specification and prototyping. Its generality means that a dialogue designer is free to make use of the dialogue state to control such aspects as help and error-handling. However, explicitly coding them into the dialogue structure can obscure the meaning of the specification. One solution is to add standard, built-in ways of dealing with such issues, such as the "pervasive" states of [Olsen 84] or "diversions" [Wartik & Pyster 83]. This sort of approach would lead to new attributes for events, for example, help, errors, levels and escapes. This in turn would mean enhancing the SPI interpreter to deal with the new attributes.

Such changes would also offer a way of introducing adaptability, since events could operate in different modes, using different attributes, depending on the required representation of the interface.

Extending eventISL offers considerable scope for making dialogue specification easier, but at the cost of complicating the very simple Read-Eval-Print model of dialogue at the heart of SPI. It remains to be seen how these conflicting requirements can best be balanced.

Other work primarily involving eventISL would be to experiment with embedding it in other languages. Some work has started on transforming me too specifications in Ada* [Clark 86]; it would be worth exploring how SPI could be set into an Ada environment, especially since Ada offers concurrent execution of tasks.

7.2.3 Using the object-oriented paradigm

The combination of encapsulation and inheritance found in object-oriented programming [Goldberg & Robson 83] [Cox 86] seems to offer a particularly

* Ada is a trademark of the U.S. Government-Ada Joint Program Office

powerful way of constructing systems. It is natural, therefore, to consider how systems might be specified in this style as well. Already, work has begun which extends me too in this direction [Minkowitz & Henderson 86]. In chapter 2, we saw that several ways of applying the object-oriented paradigm to dialogue specification are being developed, and we would like to explore how it might be incorporated into SPI.

7.2.4 Industrialising SPI

Based on experience with SPI, we believe it to be a useful, practical way of specifying and prototyping dialogues. Until tested in the world of industrial software development, this is merely a subjective opinion. Logically, the next step for SPI should be case studies based on more realistic use of the tools and techniques.

Experience with SPI has already suggested a number of improvements or additions to the tools: ways of handling standard interaction techniques such as menus and windows, structure editors for eventCSP and eventISL, process and event libraries, a more flexible outer shell with better debugging and on-the-fly modification facilities,... and so on. Better facilities for input parsing along the lines of Language-By-Example [Johnson 85] are also required. A debugger capable of handling CSP would be an asset [de Francesco et al 85], as would the ability to execute incomplete specifications [Zave & Schell 86].

Another issue not yet fully explored is how best the SPI method and tools can be used in the context of a software project. Assuming the separability of the interface from the functionality, SPI would seem to offer a useful communication tool between the two groups of designers involved. This needs to be tested in practice, as well as the underlying assumption that the languages are simple enough for use by human factors personnel unfamiliar with formal notations.

7.3 Conclusions

In SPI, we have presented an architecture for dialogues, an architecture supported by languages and tools that enable designers to specify and prototype

human-computer dialogues.

This architecture separates the sequence of events in a dialogue from the state transformational nature of those events. It was derived after experimentation with both stream-based and state-based approaches. As a result, it was found that this separation is a good way of defining dialogues, retaining both the aspects of structure and effect but not allowing either to obscure the other.

The use of CSP gives us a formal, expressive, succinct and powerful notation for specifying dialogue structure. CSP was, in many ways, a "natural" choice as a notation for expressing dialogue structure. In its earlier form, it was used because channels were an appropriate way to specify input and output streams in stream-handling operations. In its later form, processes are specifically intended to describe possible sequences of events, which is exactly what we required of a notation for dialogue structure.

There are other notations for such an event-based approach, eg. [Gorski 85] [Avrunin et al 86]; there are other notations for considering sequences of events, eg. LUCID [Wadge & Ashcroft 85]; and there are other notations for expressing concurrency, such as temporal logic [Manna & Pneuli 81], CCS [Milner 85], NIL [Strom & Yemini 85], Petri nets [Thiagarajan 85], occam [INMOS 84] and other CSP derivatives [Haase 85]. Any of these may well have proved suitable, but CSP had the advantages of familiarity, an easily implementable formal notation, our earlier experiments using streamCSP and the reported experience of others who found CSP a useful design tool [Hull & McKeag 84].

In CSP, the parallel operator (||) offers considerable scope for innovation in specifying dialogues. For example, we can easily specify concurrent input. Alternatively, it can be used to separate and synchronise the activities of related processes, thus allowing the decomposition of a dialogue into sub-dialogues. Such decomposition is illustrated in the specification of a syntax-directed editor in [Alexander 86], and is a well-known and much-advocated technique for managing complexity in software development.

Because we have been concerned with the syntactic layer of dialogues, little has been said about SPI's lexical capabilities. As stated in chapter 2, this layer covers a number of issues, such as primitive device handling (mouse, screen, keyboard, external sensor, ...), token representation and analysis, screen layout and interaction techniques (menus, forms, windows, dials, ...). It could be argued that much of this level is best defined using a traditional programming language or a specialised notation [Green 85], since it involves low-level device handling. It is true that the me too version of SPI is not particularly appropriate for the very detailed level of device handling, key presses, icon drawing etc, largely due to the nature of me too, since it was not designed to deal with such issues. Embedding SPI in C is more appropriate for this layer and offers the opportunity to handle all aspects of user interface design.

SPI does not directly address layout issues, since much of this can be specified in separate me too components and the remainder is concerned with device handling. [Rowles 86] describes a functional layout language embedded in me too which enables text-based screens to be designed and saved in a screen dictionary.

The me too-based implementation of SPI provides a specification and prototyping environment for dialogues. The combination of SPI and me too allows a software designer to formally specify and prototype most aspects of an interactive system: its functionality, the structure of its interactions with users, and input and output formats. However, this version of SPI can also be seen as a prototype in its own right, limited in its functionality (as outlined in the previous section) and particularly in its performance. Having followed the me too method in designing SPI, the next step was to use the design to implement the tools in a more conventional way.

The first phase of this implementation has been completed, offering SPI under UNIX and embedding eventISL in C. The resulting system, while not yet a fully-fledged production-quality tool, has yielded much improved performance and presentation, and now offers an implementation environment for dialogues.

Implementing SPI in this way has demonstrated that there are no major difficulties left to be resolved.

In summary, SPI has achieved its original goals; its languages are formal and executable, and it offers an integrated technique for specifying and prototyping human-computer dialogues, supported by the necessary tools. Work on related areas remains, but we believe the SPI's architecture and languages are more than an adequate foundation for that work.

APPENDIX 1

me too notation

In chapter 2, the me too method for specification and rapid prototyping was introduced. This appendix describes the me too notation that is used throughout the thesis so that the reader can understand the specifications that are given. This appendix is intended only as an outline of me too; further details can be found in the me too manual [Henderson et al 85].

A1.1 LispKit notation

Since me too is embedded in the functional language LispKit [Henderson 80], all the LispKit notation is imported into me too. This section gives a summary of LispKit notation available in me too.

arithmetic: x+y x-y x*y x/y x rem y

boolean:
b1 and b2
b1 or b2
not b
e1=e2
x<=y (less than or equal)

conditional: if b then e1 else e2

lists:

list(a,b,d)	creates the list (a b d)
head(l)	extracts the first list element
tail(l)	returns all but the first list element
cons(x,l)	adds element x at the start of list l
append(l1,l2)	concatenates lists l1 and l2

function application:

fn(e1,...,ek)

returns value of the function named fn applied to the arguments ei.

Application can be nested to any depth. For example, if

double(x) ≡ x * 2

then

double(5) = 10
double(double(1+2)) = 12

local declarations:

```
let n1 = e1
    ...
    nk = ek
in
    e
```

returns the value of e, evaluated in a context enriched by binding the names ni to the values ei

```
letrec n1 = e1
       ...
       nk = ek
in
    e
```

recursive version of the above, so that the ni may be used within the expressions ei.

The functionality (or type) of a function is given by

f: T1 x T2 x ... x Tk -> Tk+1

where f is a function with arguments of type T1,...,Tk and a result of type Tk+1.

There are strict and lazy versions of LispKit available. In most me too specifications, the evaluation strategy is immaterial; however, it should be noted that streamCSP relies on lazy evaluation of the input and output channels in order to handle input/output processing.

A1.2 Sets

enumeration: { e1, ..., ek }

with {} or Ø for the empty set

basic set operations:

s1 ∪ s2	union
s1 ∩ s2	intersection
s1 - s2	difference
e ∈ s	member
s1 ⊆ s2	subset

cardinality (size): card s

selection: the S

selects the member of the singleton set S. The result is undefined if S has more than one member.

distributed union: union S

where S is a set of sets. For example, if A = {1,2}, B = {2,5}, C = {3,1}
and S = {A,B,C}
then
union S = {5,1,3,2}

set construction: { e | n ← S }

constructs a new set by taking each element from the set S, naming it n and building a new element using e (where e is an expression involving n). "n ← S" is called a generator clause.

If S is {1,2,3}, then { x+1 | x ← S } is {2,3,4}

{ e | n ← S ; b }

as above, except that the elements of S are tested using the predicate (or "filter" b before being used to build the new set. Elements not satisfying the predicate are not used. "n ← S ; b" is also a generator clause.

With S as above, { x+1 | x ← S; x<3 } is {2,3}

{ e | g1; ...; gk }

the most general form of set construction, where each gi is a generator clause (with or without filter).

reduction: n/e S

collapses the set S into a single element with the same type as e, using binary function with name n. If S is the set { e1, ..., ek } then

n/e S = n(e1,...n(ek,e)...)

For example,

+/0 {3,4,5} = 12

U/0 = union

An object is declared to be a set by

obj = set(T)

where T is some type. All elements of a set are of the same type.

A1.3 Relations

A binary relation is a set of pairs. Since it is a set, all the set operations given in the previous section may be used with binary relations.

In what follows, we assume that x = { (a,1), (b,2), (a,3), (c,4) }

enumeration: { (e1,e2), ..., (ek,ek+1) }

relation operations:

dom r

returns the set of elements in the domain of r, so dom x = {a,b,c}

ran r

returns the set of elements in the range of r, so ran x = {1,2,3,4}

r dr S

returns a new relation, containing pairs from r whose domain element occurs in the set S, so x dr {a} = { (a,1), (a,3) }

r ds S

returns a new relation, containing pairs from r whose domain element does not occur in the set S, so x ds {a} = { (b,2), (c,4) }

An object is declared to be a binary relation by

obj = rel(T1,T2)

where domain elements are of type T1 and range elements are of type T2.

A1.4 Finite functions

A finite function is a binary relation with unique entries in the domain, that is, if F is a finite function then

card F = card dom F

In VDM terminology, this type is known as a "map". Since it is a binary relation, all the operations given in the previous section may be used with finite functions.

enumeration: { e1→e2, ..., ek→ek+1 }

construction: { n→e | n ← S }

constructs a finite function by taking each element from S (the domain), naming it n and computing the corresponding range element as e (where e is an expression involving n).

Thus if S is {1,5,7}

then { n→n+1 | n ← S } is { 1→2, 5→6, 7→8 }

{ n→e | n ← S ; b }

as above, but the filter b is applied to the elements in S.

application: f[e]

if the value of e appears in the domain of f, the result is the corresponding range element. Otherwise the result is undefined.

If f = {1→red, 7→blue, 2→green}

f[7] = blue
f[6] is undefined

f[e1]e2

as above, except that if e1 does not occur in the domain of f, the default expression e2 is returned as the result of the application. With f as above,

f[1]purple = red
f[6]purple = purple

override: f1 ⊕ f2

creates a new finite function whose domain contains the domain elements of f1 and f2. If an element occurs in the domains of both f1 and f2, the new range element is taken from f2 (hence f2 overrides f1). Otherwise the new finite function contains all the pairs in f1 together with all the pairs in f2.

If f is as above and f' = { 3→pink, 7→yellow } then

f ⊕ f' = { 1→red, 2→green, 3→pink, 7→yellow }
f' ⊕ f = { 1→red, 2→green, 3→pink, 7→blue }

An object is declared to be of type finite function by

obj = ff(T1,T2)

where domain elements are of type T1 and range elements are of type T2.

A1.5 Sequences

enumeration: < e1, ..., ek >
with empty sequence <> or nil

or { 1→e1, ..., k→ek }

since sequences can be regarded as finite functions mapping integers (the position in the sequence) to sequence elements,

concatenation (append): q1 ^ q2

returns a sequence starting with all the elements of q1 followed by all the elements of q2.

cons: cons(e,q)

returns a sequence with first element e followed by the elements of q.

sequence operations:

head(q)	
tail(q)	
len q	length of sequence
elems q	set of elements in sequence
inds q	indices of sequence
q[x]	selects element at position x

If we define Q = <c,a,b,d,f,a> then

head(Q) = c
tail(Q) = <a,b,d,f,a>
len Q = 6
elems Q = {a,f,d,b,c}
inds Q = {1,2,3,4,5,6}
Q[5] = f

override: q ⊕ { x1→e1, ..., xn→en }

returns a sequence which is the same as q, except that expressions ei are now in positions xi. With Q as defined above,

Q ⊕ { 3→x, 1→y} = <y,a,x,d,f,a>

distributed concatenation: conc Q

where Q is a sequence of sequences.

If Q = < <a,b>, <b,d>, <c> >
then
conc Q = <a,b,b,d,c>

construction: < e | n ← q >

constructs a new sequence by taking each element from the sequence q, naming it n and building a new element e (where e is an expression involving n). As with set construction, "n←q" is called a generator clause.

For example, < x*x | x ← <2,-1,3> > is <4,1,9>

< e | n ← q ; b >

as above, except that only the elements of q which satisfy the predicate (or "filter") b are used to build the new sequence. "n ← q ; b" is also a generator

clause (with filter).

<e | g1; ...; gk >

is the general form of sequence construction, where each gi is a generator clause (with or without filter).

An object is declared to be a sequence by

obj = seq(T)

where T is some type. All elements of a sequence are of the same type.

A1.6 Tuples

A tuple is an ordered group of elements which may be of different types.

enumeration: (e1,...,ek)

constructs a k-tuple

selection:

first(t)	second(t)
third(t)	fourth(t)
fifth(t)	

are the only operations available for tuples in me too. They select the appropriate entry of a tuple.

patterns: (n1,...,nk) = e

where e evaluates to a k-tuple. This is an alternative (non-standard) notation for selecting and naming components of a tuple.

If x is the tuple (1, fred, <3,7,2>) and a local declaration is made

```
letrec (a,b,c) =x
in E
```

this is equivalent to

```
let  a = first(x)
     b = second(x)
     c = third(x)
in E
```

Thus, in E, a=1, b=fred and c=<3,7,2>.

An object is declared to be a tuple by

obj = tuple(T1,...,Tk)

where the Ti may be different types.

A1.7 Constants

Strictly, all constants used in me too specifications should be quoted, eg.

F["red"]

but where the intention is clear, constants are not quoted. As a further aid, all atoms given entirely in uppercase are deemed to be constants.

A1.8 Types

In addition to the individual type declarations shown above, an object may be declared as being of type T1 or of type T2 by

obj = T1 ∪ T2

eg Flag = Int ∪ Boolean

Composite types can be declared as

obj = T1 X T2

so that an object of this type has a component of type T1 and a component of type T2.

APPENDIX 2

Specification of logon example

This component supplies operations to support the checking of users and passwords for a logon dialogue.

Objects

The system maintains a table of users and their passwords. This is represented as a finite function, mapping each user name to the appropriate password:

UserDb = ff(UserName,Password)
UserName, Password = Atom

Operations

Two operations are supplied. The first checks that the given user name is registered in the table; the second checks the supplied password against that in the table.

registered : UserDb x UserName -> Boolean
validpwd : UserDb x UserName x Password -> Boolean

where the operations are defined as follows:

registered(udb,u) ≡ u ∈ dom(udb)
validpwd(udb,u,pw) ≡ pw = udb[u]

APPENDIX 3

Specification of decision table example

This component supplies operations on a decision table containing questions, answers and decisions.

Objects

The table is represented by an n-ary tree, with questions in composite nodes, and branches labelled with possible answers.

```
Tree = Decision U ff(Question,ff(Answer,Tree))
Decision, Question, Answer = Atom
QA-pair = tuple(Question,Answer)
```

Operations

Is-decision checks to see if the tree has been reduced to a single node, ie. a decision. The next question to be asked is returned by the operation question and prune reduces the tree according to the answer given to the question.

```
is-decision : Tree -> Boolean
get-q : QA-pair -> Question
get-a : QA-pair -> Answer
question : Tree -> Question
prune : Tree x Question x Answer -> Tree
```

```
is-decision (t) ≡ atom(t)

get-q(q&a) ≡ first(q&a)

get-a(q&a) ≡ second(q&a)

question (t) ≡ the dom(t)

prune (t,q,a) ≡
        let pruned = { (a',prune(t',q,a)) | (a',t') ← t[question(t)] }
        in
            if is-decision(t)
            then t
            else if q = question(t)
            then t[q][a]
            else { question(t)→pruned }
```

APPENDIX 4

Rewrite rules for streamCSP

StreamCSP is implemented as a language embedded in me too. It is translated by a preprocessor from its S-expression form into standard me too which can then be compiled as usual and run as a prototype. The preprocessor systematically rewrites terms in the source notation using a set of rewrite rules until no more rules can be applied [Finn 84]. This appendix describes the rewrite rules used by the preprocessor for streamCSP; they are based on rules originally devised by Simon Jones of the University of Stirling.

As indicated by the example in §3.1, the translation transforms the process function from having functionality.

(user-state -> user-state)

to having the functionality

(user-state -> (in -> (out x user-state x in)))

ie. (user-state -> runnable-process)

as required by the ProtoKit "run" command which is used to execute interactive prototypes.

In the rules that follow, each is in the form of a 3-list:

```
( set-of-bound-variables
  term-to-be-written
  result-after-rewriting )
```

The first DODEFS rule specifies the first term to be rewritten, then the remaining DODEFS rules deal with individual processes, rewriting them as me too operations with the appropriate functionality. Non-process functions are left unchanged. The final rule enables internal operations to be added to the specification at the end.

```
( ( e l ) ( processes e . l ) ( letrec e DODEFS l ) )

( ( pid iv l body )
  ( DODEFS ( ( pid process iv body ) . l ) )
  ( ( pid lambda iv ( lambda ( kb ) ( DOBODY body ) ) )
                                          DODEFS l ) )
```

```
( ( id other l )
  ( DODEFS ( ( id . other ) . l ) )
  ( ( id . other ) DODEFS l ) )

( () (DODEFS NIL) ADDFUNl )
```

As a result of the DODEFS rule, each process function body is flagged by DOBODY. The DOBODY rules recognise and expand the streamCSP notation. Note that the rule for input (c?x->P) forces the evaluation of each input as it is requested in order to ensure the correct interleaving of input and output.

```
( ( c x p )
  ( DOBODY ( c ? x → p ) )
  ( let
    ( let ( if ( atom x ) the-rest the-rest )
      ( the-rest DOBODY p ) )
    ( x head kb )
    ( kb tail kb ) ) )

( ( c e p )
  ( DOBODY ( c ! e → p ) )
  ( letrec
    ( list ( cons e *os ) *ov *is )
    ( ( *os *ov *is ) DOBODY p ) ) )

( ( oe ) ( DOBODY ( return oe ) ) ( list nil oe kb ) )
```

When calling another process, only the input state is explicitly given. This rule adds the implicit input stream (kb) to the call.

```
( ( pid' ie )
  ( DOBODY ( pid' . ie ) )
  ( ( pid' . ie ) kb ) )
```

Sequential composition is implemented by an internal operation called SEQ (see below). This rule translates the ; construct into a call on SEQ, adding the input stream parameter as for a process call.

```
( ( pidl pid2 ie )
  ( DOBODY (( pidl ; pid2 ) . ie ) )
    ( ( ( SEQ pidl pid2 ) . ie ) kb ) )
```

The following rules carry the DOBODY translations through the normal me too constructs.

```
( ( c pl p2 )
  ( DOBODY ( if c pl p2 ) )
  ( if c ( DOBODY pl ) ( DOBODY p2 ) ) )
```

```
( ( p l )
  ( DOBODY ( let p . l ) )
  ( let ( DOBODY p ) . l ) )

( ( p l )
  ( DOBODY ( letrec p . l ) )
  ( letrec ( DOBODY p ) . l ) )
```

The rules below add the SEQ operation required for sequential composition and a simple pattern-matching operation.

```
( NIL
  ADDFUN1
  ( ( SEQ lambda ( P Q )
      ( lambda ( st )
        ( lambda ( kb )
            ( letrec
                ( list ( append out1 out2 ) st2 kb2 )
              ( ( out1 st1 kb1 ) ( P st ) kb )
              ( ( out2 st2 kb2 ) ( Q st1 ) kb1 ) ) ) ) )
                                        . ADDFUN2 ) )

( NIL
  ADDFUN2
  ( ( matches
      lambda
      ( x t )
      ( if
        ( atom t )
        ( or ( eq t ( quote ANYTHING ) ) ( eq x t ) )
      ( if
        ( atom x ) false
      ( if
        ( eq ( head x ) ( head t ) )
        ( eq ( length ( tail x ) ) ( length ( tail t ) ) )
        false ) ) ) ) )
```

APPENDIX 5

Specification of ECS-state interpreter

Objects

ECS-state (see chapter 3)
in and out are the input and output streams respectively

Operations

The interpreter is implemented by a group of me too stream-handling operations. The outermost operation has type

ECS : AppState –> (in –> (out x AppState x in))

The remaining processes are:

ecs1 : ECS-state –> (in –> (out x ECS-state x in))
ecs2 : ECS-state –> (in –> (out x ECS-State x in))

```
ECS(as)(kb) ≡ ecs1(start(as))(kb)

ecs1(st)(kb) ≡
    if st[TERMINATE]false
    then list(nil,st[DB],kb)
    else if st[INPUT-REQD]false
    then
        let inp = head(kb)
            kb = tail(kb)
        in ecs2(edit(st ds {INPUT_REQD},inp))(kb)
    else ecs2(st)(kb)

ecs2(st)(kb) ≡
    let newst = compute(st)
    in
        if OUTPUT∈dom(newst)
        then let (out,newst',kb') =
                        ecs1(newst ds {OUTPUT})(kb)
             in list(show(newst)^out,newst',kb')
        else ecs1(newst)(kb)

show(st) ≡ st[OUTPUT]<>

edit(st,i) ≡ st ⊕ {INPUT→i}
```

APPENDIX 6

Syntax definitions for eventCSP

The first section gives the definition of the abstract syntax given in chapter 3. The following sections define the concrete syntax by showing how each construct from the abstract syntax is represented. Note that there is no explicit representation for the "abort" process.

A6.1 Abstract syntax

A process is defined as follows:

if e, e1,...,en are events and P, P1,...,Pn are processes, then the following are also processes

(e → P)	- (prefix) engage in event e then behave like P
(e1 → P1	- (choice) engage in e1
[] e2 → P2	- then behave like P1, or engage in e2
[] ...	and behave like P2, etc
[] en → Pn)	
P1 ; P2	- (sequence) P1 followed by P2 if P1 terminates
P1 || P2	- (parallel) P1 in parallel with P2
l : P	- (label) label P with l
skip	- successful termination
abort	- no further interaction

A6.2 Concrete syntax for me too

This version adopts the S-expression form used in the concrete syntax of me too. For ease of reading, this definition shows constant strings thus: 'seq

(e → P)	('ev e P)
P [] Q	('alt P Q)
P ; Q	('seq P Q)
P	('call P)
P || Q synchronised on {x,y,z}	('par P Q (SET x y z))
l : P	('label l P)
skip	'skip
Pname = Q	(Pname Q)

A6.2 Concrete syntax for C

(e → P)	(e -> P)
P [] Q	(P ^ Q)
P ; Q	(P ; Q)
P || Q synchronised on {x,y,z}	(P || Q {x y z})
l : P	(l : P)
skip	skip
Pname = Q	Pname ≐ Q

APPENDIX 7

Syntax definitions for eventISL

The first section below defines the abstract syntax for eventISL. EventISL is an embedded language and so its concrete syntax is heavily influenced by its host language. Thus far it has been embedded in me too and C. The remaining sections in the appendix give the concrete syntax for these versions.

In the following sections, the syntax is defined using the following notation:

<XYZ>	XYZ names a non-terminal symbol
{XYZ}+	XYZ repeated one or more times
{XYZ}*	XYZ repeated none or more times
[XYZ]	XYZ is optional
ABC | XYZ	ABC or XYZ

A7.1 Abstract syntax

```
Dialogue      ::= <ProcessOp> {<EventOp>}*
ProcessOp     ::= process <ProcessName> = <DlgExpr>
EventOp       ::= event <EventName> = <DlgExpr>

DlgExpr       ::= <UseExpr> | <AttrExpr>
UseExpr       ::= use {UseVar}+ in <AttrExpr>
AttrExpr      ::= <LetExpr> | <AttrList>

LetExpr       ::= let {LetPair}+ in <AttrExpr>
LetPair       ::= <LetVar> = <Expr>

AttrList      ::= empty | <AttrList> <Attr>
Attr          ::= when <BoolExp> |
                  prompt <BoolExp> |
                  out <Expr> |
                  retain {<ApprVar>}+ |
                  remove {<RemVar>}+ |
                  <AppVar> = <Expr>
Expr                any valid expression in host language
LetVar, UseVar, AppVar, RemVar, EventName, ProcessName
                    any valid identifier in host language
```

Note that this does not rule out the use of when in a process specification; however any condition operation generated is not used.

A7.2 Concrete syntax for me too

This version adopts the S-expression form used in the concrete syntax of me too. For ease of reading, this definition shows constant strings thus: 'event

```
Dialogue        ::= ( <ProcessOp> . <EventOpList> )
ProcessOp       ::= ( 'process <ProcessName> <DlgExpr> )
EventOpList     ::= NIL | ( <EventOp> . <EventOpList> )
EventOp         ::= ( 'event <EventName> <DlgExpr> )

DlgExpr         ::= <UserExpr> | <AttrExpr>
UseExpr         ::= ( 'use <UseVarList> <AttrExpr> )
UseVarList      ::= NIL | ( <UseVar> . <UseVarList> )
AttrExpr        ::= <LetExpr> | ( 'attrs . <AttrList> )

LetExpr         ::= ( 'let <LetPairList> <AttrExpr> )
LetPairList     ::= NIL | ( <LetPair> . <LetPairList> )
LetPair         ::= ( <LetVar> . <Expr> )

AttrList        ::= NIL | ( <Attr> . <AttrList> )
Attr            ::= ( 'when <BoolExp> ) |
                    ( 'prompt <BoolExp> ) |
                    ( 'out <Expr> ) |
                    ( 'retain <AppVarList> ) |
                    ( 'remove <AppVarList> ) |
                    ( <AppVar> <Expr> )
AppVarList      ::= NIL | ( <AppVar> . <AppVarList> )
```

For example,

```
( 'event user-answer
        ( 'use (dt qu input)
        ( 'attrs
          ( dt ( prune dt qu input ) ))))
```

or

```
( 'event ask-question
        ( 'use (dt)
        ( 'let
          ( 'attrs
              ( 'out qu )
              ( 'retain qu ) )
          ( qu question dt ) )))
```

A7.3 Concrete syntax for C

This version of eventISL is strongly influenced by the imperative nature of C. The most significant difference reflects the fact that all application-specific data is declared as variables in the eventISL specification. As a result none of the state manipulation expressions are included in this version of the language.

```
Dialogue        ::= DIALOGUE <Text> <ProcessOp>
                        {<EventOp>}*
ProcessOp       ::= PROCESS <ProcessName> <DlgExpr>
EventOp         ::= EVENT <EventName> [<When>] <DlgExpr>

DlgExpr         ::= PROMPT <Text> <DlgExpr> |
                    OUT <Text> <DlgExpr> |
                    TEXT <Text> <DlgExpr> | nothing
When            ::= WHEN <Text>

Text            ::= {Chunk}+
Chunk           the words "input", "db", or any other text except
                one of the keywords above.
```

In other words, any unrecognised text (in the right places) is ignored by the syntax and assumed to be valid C code. For example,

```
EVENT user_answer
        TEXT dt = prune(dt,qu,input) ;

EVENT ask_question
        TEXT strcpy(qu,(char *)question(dt));
        PROMPT TRUE
        OUT "\n%s ? ",qu
```

APPENDIX 8

Specification of SCHOLAR example

SCHOLAR is a CAI system described in [Carbonnell 70]. The original system exhibited a number of distinctive features, of which two are modelled by this component. One is the style of interaction: Carbonnell coined the phrase "mixed initiative" to describe this style, where either partner (student or system) can take the initiative and ask questions of the other. The other is that questions are derived from information in the database, instead of being stored directly. This specification describes a simplified version of this feature, in that it does not include any inference mechanism which would allow the system to deduce, for example, that if Lima is the capital of Peru and Peru is in South America, then Lima is in South America.

A8.1 SCHOLAR

Objects

ScholarDb = FDM-database
UserInp, Answer = seq(Atom)
UserQu = tuple({QU},Entity,TableName) U tuple({QU},Entity)
SchQu = tuple(Entity,TableName,{?})
Reply = { RIGHT, WRONG }

and a constant Null Qu = <>

For definitions of FDM-database, Entity and TableName, see A8.3 below (specification of FDM).

Operations

Initdb and register set up and update (respectively) a table in the database which records questions that have already been asked. Pickq constructs a question from the information in the database, avoiding questions already used; a question is made up from an entry and the name of a table in which it appears, eg "Bolivia has-capital?"

```
initdb : ScholarDb -> ScholarDb
register : ScholarDb x SchQu -> ScholarDb
pickq : ScholarDb -> SchQu
```

The student may also ask questions. These have the general form

QU param1 param2

where param1 is mandatory and names an entity, param2 is optional and names an attribute (ie. a TableName). If the full format is used, a single answer is given; if the short form is used, all information about the entity is returned; for example

```
"QU Peru has-capital" yields "Lima"
"QU Bolivia"          yields "is-a country;
                              is-in South America"
```

The next three operations all deal with user questions: is-question checks for the QU keyword, query answers the question, using all-about if the short question form is given.

```
is-question : UserInp -> Boolean
query : ScholarDb x UserQu -> Answer
all-about : ScholarDb x Entity -> set(Answer)
```

When a student answers a question, check compares the answer with the facts in the database.

```
check : ScholarDb x SchQu x UserInp -> Reply
```

The SCHOLAR database holds both the subject information and some meta-information for system control. The system maintains a dictionary of tables in the CONTAINS table, flagging subject tables as "T" and meta-tables as "MT". The TYPE table is a dictionary of entities and their types. The ASKED table notes which questions have been asked for each entry in the subject information.

```
init(db) ≡ dbadd( dbdel(db,ASKED),
                  ASKED,
                  fnadds(Ø,dom(get(db,TYPE)),Ø) )

register (db,q) ≡ dbadd( db,
                         ASKED,
                         fnadd(get(db,ASKED),first(q),second(q)) )

pickq (db) ≡
        let ETpairs = { (e,t,?) | e ← used(db);
                                  t ← tables-in(db);
                                  e ∈ dom(get(db,t))
                                  and t ∉ asked-about(db,e) }
        in
```

```
        if ETpairs = ∅
        then "No more questions"
        else any(ETpairs)
                where any(s) = head(s)

is-question (ui) ≡ not atom(ui) and first(ui) = QU

query (db,uq) ≡
        if length(uq) = 2
        then all-about(db,second(uq))
        else apply(get(db,third(uq)),{second(uq)})

all-about (db,e) ≡ { <e,t,apply(get(db,t),{e})> |
                        t ← tables-in(db); e ∈ dom(get(db,t)) }

check (db,q,a) ≡
        let SCHans = apply(get(db,2(q)),{1(q)})
        in if a ∈ SCHans then RIGHT else WRONG
```

A8.2 Subsidiary operations

```
tables-in : ScholarDb -> set(TableName)
used : ScholarDb -> set(Entity)
asked-about : ScholarDb x Entiry -> set(TableName)

tables-in (db) ≡ apply(get(db,CONTAINS),{T})

used (db) ≡ union { dom(get(db,t)) | t ← tables-in(db) }

asked-about (db,e) ≡ apply(get(db,ASKED),{e})
```

For descriptions of get, dbadd, dbdel, dom, apply, fnadds and fnadd, see section below.

A8.3 Specification of FDM

This component provides some operations associated with the Functional Data Model [Gray 84]. The database can be regarded as a collection of named tables. We give only the me too model of FDM; ie. the objects and an informal description of the operations.

Objects

```
FDM-database = ff(TableName,Table)
Table = ff(Index,EntitySet)
Index = Entity
EntitySet = set(Entity)
TableName, Entity = atom
```

Operations

Get retrieves a named table from the database. Apply returns all entities in the rows indicated by the set of index entities given to it.

```
get : FDM-database x TableName -> Table
apply : Table x set(Entity) -> set(Entity)
```

Dom returns all entities in the first column of a table; ran returns all entities appearing in the second column of a table.

```
dom : Table -> set(Entity)
ran : Table -> set(Entity)
```

The database can be updated by adding or deleting named tables.

```
dbadd : FDM-database x TableName x Table -> FDM-database
dbdel : FDM-database x TableName -> FDM-database
```

The next operations create a single row, then add or remove its contents from the table. For addition, this may create a new row (if the index entity was not already present in the first column) or add to the EntitySet of an existing row. For removal, removing the last member of an EntitySet in a row deletes the row from the table.

```
fnadd : Table x Entity x Entity -> Table
fndel : Table x Entity x Entity -> Table
```

These operations are similar but create a number of rows (the cross-product of the two sets supplied) instead of just one.

```
fnadds : Table x set(Entity) x set(Entity) -> Table
fndels : Table x set(Entity) x set(Entity) -> Table
```

APPENDIX 9

Specification of forms example

A form is a sequence of single field entries, each of which can solicit one input from the user. The form structure allows the designer to specify default values, help texts, mandatory fields and inter-field dependencies. These are among a number of facilities recommended in [Gehani 83].

Objects

```
FormDb     = ff(FormName,Form)
Form       = seq(Field)
Field      = tuple(fn:FieldName,fa:FieldAttr)
FieldAttr  = tuple(def:FieldValue,val:FieldValue,
                   pos:Position,help:Text,
                   requ:Boolean,depd:set(FieldName))
FieldValue, FormName, FieldName = Text
Position   = tuple(x:Int,y:Int)
Text       = seq(Atom)
```

For compactness of specification, we have associated identifiers with each element in a tuple. These act as "selectors", so that an element in a tuple is extracted as

this-form.fn

or any-field.fa.pos.x

to yield the form name and x coordinate, respectively. This is not standard me too notation, but does make the specification shorter and easier to read [Clark 86].

Constants

xstart, ystart : define an initial position on the screen

Operations

Creating fields

mkfield : FieldName x FieldAttr -> Field

mkfield(fn,fa) ≡ list(fn,fa)

Extracting form and field names

```
forms : FormDb -> set(FormName)
fields : Form -> seq(FieldName)
```

```
forms(fdb) ≡ dom(fdb)
fields(f) ≡ < fld.fn | fld ← f >
```

Database query and update

```
get-form : FormDb x FormName -> Form
update : FormDb x FormName x Form -> FormDb
form-exists : FormDb x FormName -> Boolean

get-form(fdb,fb) ≡ fdb[fn]
update(fdb,fn,f) ≡ fdb ⊕ { fn → f }
form-exists(fdb,fn) ≡ fn ∈ forms(fdb)
```

Updating and checking forms

```
clear : Form -> Form
remove-field : seq(Field) x Field -> seq(Field)
enter : Form x FieldName x FieldValue -> Form
in-form : Form x FieldName -> Boolean
defaults : Form -> Form
not-complete : Form -> seq(Field)
newvalue : Field x FieldValue -> Field

clear(f) ≡ < newvalue(fld,<>) | fld ← f >

remove-field(flds,fld) =
        < fld' | fld' ← flds; fld.fn ≠ fld'.fn>

enter(f,fdn,fdv) ≡
        < if fdn = fld.fn then newvalue(fld,fdv)
          else fld | fld ← f >

in-form(f,fdn) ≡ fdn ∈ elems fields(f)

defaults(f) ≡
        < if fld.fa.def ≠ <> and fld.fa.val = <>
          then newvalue(fld,fld.fa.def)
          else fld | fld ← f >

not-complete(f) ≡
        let given = elems < fld.fn | fld ← f;
                                fld.fa.val = <> >
        in
            < fld | fld ← f; fld.fa.val = <>
                    and (fld.fa.requ
                        or given ∩ fld.fa.depd ≠ ∅) >

newvalue(fld,v) ≡
        let a = fld.fa
        in
            mkfield(fld.fn,list(a.def,v,a.pos,a.help,a.requ,a.depd))
```

Extracting field attributes

```
get-name : Field -> FieldName
get-value : Field -> FieldValue
get-help : Field -> Text

get-name(fld) = fld.fn
get-value(fld) = fld.fa.val
get-help(fld) = fld.fa.help
```

Displaying forms

```
form-menu : FormDb -> Text
display-form : FormDb x FormName -> Text
display-title : FormName -> Text
display-fields : Form -> Text
display-field : Field -> Text
vstart : Field -> Position

form-menu(fdb) ≡
        let m = sort(forms(fdb))
        in
            < "Forms available:" > ^ conc < nl() ^ opt | opt ← m >

display-form(fdb,fn) ≡
        let f = get-form(fdb,fn)
        in pretty(display-title(fn) ^ display-fields(f))

display-title(fn) ≡ < curs(start,ystart), fn >

display-fields(f) ≡ conc < display-field(fld) | fld ← f >

display-field(fld) ≡
        let start = vstart(fld)
        in
            < curs(xstart+fld.fa.pos.x, ystart+fld.fa.pos.y),
              fld.fn, curs(start.x,start.y), fld.fa.val >

vstart(fld) ≡
        list(xstart+fld.fa.pos.x+length(explode(fld.fn)),
             ystart+fld.fa.pos.y)
```

External operations

The forms component imports operations from a screen component. One allows the cursor to be positioned; the other creates a "newline":

```
curs : Int x Int -> Text
nl : -> Text
```

APPENDIX 10

Specification of forms dialogue

This appendix gives the full specification of the forms dialogue, including the C version.

A10.1 EventCSP specification

```
forms = ( menu →
            ( valid-form → get-form → fill-in ; forms
            [] repeat? → get-form → fill-in ; forms
            [] inv-form → error → forms )

fill-in = ( fields-left? → position → old-value
            → position → get-input →
                ( help? → fill-in
                [] skip? → fill-in
                [] cancel? → forms
                [] undo? → fill-in
                [] save? → check-form
                [] value? → update → fill-in )
          [] not-fields-left? → check-form )

check-form = ( complete? → save-form → skip
             [] not-complete? → fill-in )
```

A.10.2 EventISL specification - me too version

The AppState is:

entry index	type	used for
thisf	FormName	name of current form
lastf	FormName	name of previous form
f	Form	current form
flds	seq(Field)	fields to process
fld	Field	current field
done	seq(Field)	fields processed
fdb	FormDb	form database

The events are:

```
event menu =
use fdb in
    out form-menu(fdb)
    prompt true
```

```
event valid-form =
use fdb, input in
     when not matches(input,"REPEAT")
          and form-exists(fdb,input)
     thisf = input
     lastf = thisf
     f = clear(get-form(fdb,input))

event inv-form =
use fdb, input in
     when not ( matches(input,"REPEAT")
               or form-exists(fdb,input) )

event repeat? =
use fdb, input, lastf in
     when matches(input,"REPEAT") and form-exists(fdb,lastf)
     thisf = lastf
     f = get-form(fdb,lastf)

event get-form =
use fdb, f, thisf in
     out display-form(fdb,thisf)
     done = <>
     flds = fields(f)

event error =
     out "error: no such form"

event fields-left? =
use flds in
     when flds ≠ <>
     fld = head(flds)

event not-fields-left? =
use flds in
     when flds = <>

event position =
use fld in
     out vstart(fld)

event old-value =
use fld in
     out get-value(fld)

event get-input =
     prompt true

event skip? =
use flds, done, fld, input in
     when matches(input,"SKIP")
     flds = tail(flds)
     done = <fld> ^ done

event cancel? =
use input in
     when matches(input,"CANCEL")
     flds = <>
     done = <>
```

```
event undo? =
use flds, done, input in
     when matches(input,"UNDO")
     flds = head(done) ^ flds
     done = tail(done)

event save? =
use input in
     when matches(input,"SAVE")
     done = <>

event help? =
use input,fld in
     when matches(input,"HELP")
     out get-help(fld)

event value? =
use input in
     when not ( matches(input,"CANCEL")
                or matches(input,"UNDO")
                or matches(input,"SAVE")
                or matches(input,"SKIP")
                or matches(input,"HELP") )

event update =
use f, flds, done, fld, input in
     f = enter(f, get-name(fld), input)
     done = <fld> ^ done
     flds = remove-field(flds,fld)

event complete? =
use f in
     when not-complete(f) = <>

event not-complete? =
use f in
     let to-do = not-complete(f)
     in
          when not to-do = <>
          out "error: some required fields not given"
          flds = to-do
          done = <>

event save-form =
use fdb, f in
    flds = <>
    done = <>
    fdb = update(fdb,f)
```

A10.3 EventISL specification - C version

DIALOGUE

```
/*****
  *
  *      forms - eventISL specification
  *
  *****/

#include "form.h"
```

```
/*** externals to patterns component ***/
extern unsigned matches();

/*** externals to forms component ***/
extern FDB_PTR form_example();
extern FDB_PTR update();
extern unsigned exists_form();
extern char *form_menu();
extern char *display_form();
extern char *vstart();
extern FLDS_PTR get_form();
extern FLDS_PTR enter();
extern FLDS_PTR clear();
extern FLDS_PTR fields();
extern FLDS_PTR not_complete();
extern FLDS_PTR join();
extern FLDS_PTR copy_field();
extern FLDS_PTR remove_field();

/*** externals to screen component ***/
extern char *curs();
extern char *cls();

/*** declarations for forms application ***/
static char this_form[25],
            last_form[25];
static FDB_PTR fdb;
static FLDS_PTR f, flds, fld, done;
static char *str, *ctrl;

PROCESS forms
        TEXT fdb = form_example();

EVENT menu
        TEXT str = form_menu(fdb);
             ctrl = cls();
        OUT "%s\n\n%s\nType name of form: ",ctrl,str
        PROMPT TRUE
        TEXT free(str); free(ctrl);

EVENT valid_form
        WHEN (!matches(input,"REPEAT") && exists_form(fdb,input))
        TEXT strcpy(last_form,this_form);
        strcpy(this_form,input);
             f = clear(get_form(fdb,input));

EVENT inv_form
        WHEN !(matches(input,"REPEAT") || exists_form(fdb,input))

EVENT is_repeat
        WHEN matches(input,"REPEAT") && exists_form(fdb,last_form)
        TEXT strcpy(this_form,last_form);
             f = get_form(fld,this_form);

EVENT error
        OUT "\nerror: can't find form %s\n",input
```

```
EVENT get_form
        TEXT str = display_form(fdb,this_form);
        OUT "%s",str
        TEXT free(str);
        TEXT done = NULL; flds = fields(f);

EVENT fields_left
        WHEN flds != NULL
        TEXT fld = copy_field(flds->fn,flds->fa,NULL);

EVENT not_fields_left
        WHEN flds == NULL

EVENT position
        OUT "%s",vstart(fld)

EVENT old_value
        OUT "%s",(fld->fa)->value

EVENT get_input
        PROMPT TRUE

EVENT is_help
        WHEN matches(input"HELP")
        OUT "%s\n",(fld->fa)->help

EVENT is_skip
        WHEN matches(input,"SKIP")
        TEXT flds = remove_field(flds,fld->fn);
             done = join(fld,done);

EVENT is_undo
        WHEN matches(input,"UNDO")
        TEXT /* take the last field done ... */
             fld = copy_field(done->fn,done->fa,NULL);
             /* ... put it on the "fields-to-do" list ... */
             flds = join(fld,flds);
             /* ... and take it off the "done" list */
             done = remove_field(done,done->fn);

EVENT is_save
        WHEN matches(input,"SAVE")
        TEXT done = NULL;

EVENT is_cancel
        WHEN matches(input,"CANCEL")
        TEXT flds = done = NULL;

EVENT is_value
        WHEN !(matches(input,"HELP") || matches(input,"SKIP") ||
               matches(input,"UNDO") || matches(input,"SAVE") ||
               matches(input,"CANCEL"))

EVENT update
        TEXT enter(fld,input);
             done = join(fld,done);
             flds = remove_field(flds,fld->fn);
```

```
EVENT completed
        WHEN not_complete(f) == NULL
        TEXT free_names(done); free_names(flds);

EVENT not_completed
        WHEN not_complete(f) != NULL
        TEXT flds = not_complete(f);
             done = NULL;
        OUT  "\nerror: some mandatory fields omitted\n"

EVENT save-form
        TEXT str = display_form(this_form,f);
             ctrl = curs(0,40);
        OUT "%s%sForm saved\n\n",str.ctrl
        TEXT fdb = update(fdb,this_form,f);
             free(str);
             free(ctrl);
```

APPENDIX 11

Translating eventISL to me too

This appendix formally defines the rules which were given in chapter 5.

A11.1 EventISL syntax

The rules are defined in terms of abstract syntax for eventISL, which we repeat here for reference.

```
Dialogue        ::= <ProcessOp> {<EventOp>}*
ProcessOp       ::= process <ProcessName> = <DlgExpr>
EventOp         ::= event <EventName> = <DlgExpr>

DlgExpr         ::= <UseExpr> | <AttrExpr>
UseExpr         ::= use {UseVar}+ in <AttrExpr>
AttrExpr        ::= <LetExpr> | < AttrList>

LetExpr         ::= let {LetPair}+ in <AttrExpr>
LetPair         ::= <LetVar> = <Expr>

AttrList        ::= empty | <AttrList> <Attr>
Attr            ::= when <BoolExp> |
                    prompt <BoolExp> |
                    out <Expr> |
                    retain {<AppVar>}+ |
                    remove {<RemVar>}+ |
                    <AppVar> = <Expr>
Expr            any valid expression in host language
LetVar, UseVar, AppVar, RemVar, EventName, ProcessName
                any valid identifier in host language
```

We make use of the same syntactic notation, ie. {}* and {}+, in the rules to allow concise expression of rules for repeated constructs.

A11.2 Translation rules

This section describes that translation for each construct, using the following notation:

if e represents some eventISL text, then
C[e]
represents its translation as required for the condition operation;
A[e]
represents its translation as required for the action operation.

```
C[<Dialogue>]   =   { C[<EventOp>] }*
A[<Dialogue>]   =   A[<ProcessOp>] { A[<EventOp>] }*

A[<ProcessOp>]  =   A[<EventOp>]
```

C[<EventOp>] = <EventName>–C(dlg) = C[<DlgExpr>]
A[<EventOp>] = <EventName>–A(dlg) = A[<DlgExpr>]

The use expression is translated in the same way for both types of operation:

C[<UseExpr>] = let { C[<UseVar>] }* in C[<AttrExpr>]
A[<UseExpr>] = let { A[<UseVar>] }* in A[<AttrExpr>]

C[<UseVar>] = <UseVar> = dlg["<UseVar>"]
A[<UseVar>] = C[<UseVar>]

The translation of a let expression only affects the <AttrExpr> it contains:

C[<LetExpr>] = let { <LetPair> }+ in C[<AttrExpr>]
A[<LetExpr>] = let { <LetPair> }+ in A[<AttrExpr>]

The special names for entries in the system part of the state are treated in the same way for both types of operation.

C[input] = A[input] = IN$
C[db] = A[db] = DB$

The next group of rules all deal with the translation of an <AttrList>:

C[empty] = true
A[empty] = dlg

C[<AttrList> when <BoolExp>] = C[<AttrList>] and <BoolExp>
A[<AttrList> when <BoolExp>] = A[<AttrList>]

C[<AttrList> prompt <BoolExp>] = C[<AttrList>]
A[<AttrList> prompt <BoolExp>] = A[<AttrList>] ⊕ { IR$→ <BoolExp> }

C[<AttrList> out <Expr>] = C[<AttrList>]
A[<AttrList> out <Expr>] = A[<AttrList>] ⊕ { OUT$→ <Expr>}

C[<AttrList> retain {<AppVar>}*] = C[<AttrList>]
A[<AttrList> retain {<AppVar>}*] = A[<AttrList>] ⊕ { {A[<AppVar>]}* }

A[<AppVar>] = "<AppVar>" → <AppVar>

C[<AttrList> remove {<RemVar>}*] = C[<AttrList>]
A[<AttrList> remove {<RemVar>}*] = A[<AttrList>] ds { {A[<RemVar>]}* }

A[<RemVar>] = "<RemVar>"

C[<AttrList> <AppVar> = <Expr>] = C[<AttrList>]
A[<AttrList> <AppVar> = <Expr>] = A[<AttrList>] ⊕ { "<AppVar>"→ <Expr> }

APPENDIX 12

Translating eventISL to C

We use the following notation:

if e represents some eventISL text, then
T[e]
represents its translation.

The translation rules are illustrative in nature, showing the effects of translating each construct by example, rather than formally in terms of the syntax:

```
T[PROCESS ex E]          =    unsigned A_ex() { T[E] }
T[EVENT ex
  WHEN b
  E       ]              =    unsigned C_ex()
                                  { T[WHEN b] }
                              unsigned A_ex() {T[E]}
T[EVENT ex
  E       ]              =    unsigned A_ex() {T[E]}
T[WHEN b]                =    return b ;
T[PROMPT b]              =    _prompt = b ;
T[OUT txt]               =    sprintf(_out,txt) ;
T[TEXT txt]              =    txt
T[input]                 =    _input
T[db]                    =    _db
```

APPENDIX 13

Specification of the event manager

This appendix gives specifications of subsidiary and external operations called by the event manager. The main operations (initdlg, editdlg, nextdlg and showdlg) are specified in chapter 5.

Subsidiary operations

```
choose-event : set(EventName) x DlgState -> EventName
call-cond : EventName x DlgState -> Boolean
call-action : Name x DlgState -> DlgState

where   Name = ProcessName U EventName
        GenName = { Name-A, Name-C }

choose-event(evs,dlg) ≡
    let poss = { ev | ev ← evs; call-cond(ev,dlg) }
    in  if poss = Ø then ABORT
        else arb(poss)
```

where "arb" arbitrarily selects one member from a set

```
call-action (nm,dlg) ≡ APPLY (ev-act(nm), (dlg))

call-cond (ev, dlg)  ≡ APPLY (ev-cond(ev), (dlg))
```

where "APPLY" is a Lisp function, applying the function labelled by the name given in the first argument to the parameter list given in the second argument

```
ev-act : Name -> GenName
ev-cond : EventName -> GenName
add-suffix : Name x seq(Char) -> GenName
expand-name : seq(Atom) -> seq(Char)

ev-act(nm) ≡ add-suffix(nm, "-A")

ev-cond(en) ≡ add-suffix(en,"-C")

add-suffix(n,suf) ≡
  let n' = if atom(n) then < n, suf >
           else < head(n), ":", tail(n), suf >
  in
     implode (expand-name(n'))
```

```
expand-name(n) ≡
  if n = ◇ then n
  else explode(head(n)) ^ expand-name(tail(n))
```

where "implode" and "explode" are primitive operations that pack and unpack characters in an atom.

External operation

The event manager calls one operations provided by the eventCSP simulator, to determine when the eventCSP process being executed has finished.

```
process-end : set(EventName) → Boolean

process-end (evs) ≡
  evs = { } or ( card evs = 1 and TICK ∈ evs )
```

REFERENCES

Abramsky S, Sykes R (1985)
"SECD-m: a virtual machine for applicative programming"
in Functional Programming Languages and Computer Architecture (LNCS 201) pp.81-98
ed. J.P.Jouannaud; publ. Springer-Verlag, Berlin

Alexander H (1985)
"Formal specification and rapid prototyping techniques for human-computer interaction"
Technical Report TR.26
Dept. of Computing Science, University of Stirling

Alexander H (1986)
"SPI: specifying and prototyping interaction"
submitted to International Journal of Man-Machine Studies

Alty J L (1984)
"Use of path algebras in an interactive adaptive dialogue system"
in [INTERACT 84] pp.351-354

Alty J L, Brooks A (1985)
"Microtechnology and user-friendly systems - the CONNECT dialogue executor"
Research Report MMIGR.139
Heriot-Watt/Strathclyde MMI Unit, University of Strathclyde

Alvey Directorate (1984a)
Alvey MMI Strategy, publ. IEE, London

Alvey Directorate (1984b)
Alvey Programme Annual Report 1984, publ. IEE, London

Anderson S O (1985)
"Specification and implementation of user interfaces: Example: a file browser"
Draft Report
Dept. of Computer Science, Heriot-Watt University

Anderson S O (1986)
"Proving properties of interactive systems"
in [HCI 86] pp.402-416

Avrunin G S, Dillon L K, Wileden J C, Riddle W E (1986)
"Constrained expressions: adding analysis capabilities to design methods for concurrent software systems"
IEEE Transactions on Software Engineering SE-12, 2 pp.278-291

Backus J (1978)
"Can programming be liberated from the von Neumann style?"
Communication of the ACM 21, 8 pp.613-641

Badre A N (1984)
"Designing transitionality into the user-computer interface"
in [Salvendy 84] pp.27-34

Bailey R (1985)
"A HOPE tutorial"
Byte 10, 8 pp.235-258

Balbin I, Poole P C, Stuart C J (1985)
"On the specification and manipulation of forms"
in System Description Methodologies pp.239-252
ed. D.Teichroew, G.David

Barker P G (1984)
"MICROTEXT - a new dialogue programming language for microcomputers"
Journal of Microcomputer Applications 7, 2 pp.167-188

Barringer H, Kuiper R, Pneuli A (1985)
"A compositional temporal approach to a CSP-like language"
in Formal Models in Programming pp.207-227
ed. E.Neuhold,G.Chroust

Belkhouche B, Urban J E (1984)
"An executable specification language for abstract data types"
in Software Engineering: Practice and Experience pp.66-70
ed. E.Girard

Benbasat I, Wand Y (1984)
"A structured approach to designing human-computer dialogues"
International Journal of Man-Machine Studies 21, 2 pp.105-126

Berry D M, Wing J (1985)
"Specifying and prototyping: some thoughts on why they are successful"
in [Ehrig et al 85] pp.117-128

Bewley W L, Roberts T L, Schroit P, Verplank W L (1983)
"Human factors testing in the design of Xerox's 8010 'Star' office workstation"
in [CHI 83] pp.72-77

Bjorner D, Jones C B (1982)
Formal Specification & Software Development
publ. North-Holland, Amsterdam

Bleser T, Foley J D (1982)
"Towards specifying and evaluating the human factors of user-computer interfaces"
in [Gaithersburg 82] pp.309-314

Blum B I (1983)
"Still more about rapid prototyping"
ACM SIGSOFT Software Engineering Notes 8, 3 pp.9-11

Bobrow D, Kaplan R, Kay M, Norman D, Thompson H, Winograd T (1977)
"GUS - a frame-driven dialogue system"
Artificial Intelligence 8, 2 pp.155-174

Boehm B, Gray T E, Seewaldt T (1984)
"Prototyping vs. specifying: a multi-project experiment"
IEEE Transactions on Software Engineering SE-10, 3 pp.290-303

Bonet R, Kung A (1984)
"Structuring into subsystems: the experience of a prototyping approach"
ACM SIGSOFT Software Engineering Notes 9, 5 pp.23-27

van den Bos J, Plasmeijer M J, Hartel P H (1983)
"Input-output tools: a language facility for interactive and real-time systems"
IEEE Transactions on Software Engineering SE-9, 3 pp.247-259

Botting R J (1985)
"On prototypes vs. mockups vs. breadboards"
ACM SIGSOFT Software Engineering Notes 10, 1 p.18

Bournique R, Treu S (1985)
"Specification and implementation of variable, personalized graphical interfaces"
International Journal of Man-Machine Studies 22, 6 pp.663-684

Brooks F B (1985)
The Mythical Man-Month
publ. Addison-Wesley, Massachusetts

Brown J W (1982)
"Controlling the complexity of menu networks"
Communications of the ACM 25, 7 pp.412-418

Browne D P, Sharrat B D, Norman M A (1986)
"The formal specification of adaptive user interfaces using CLG"
in [CHI 86] pp.256-260

Bruce E (1986)
"A formal specification of a 'Prospector'-type expert system shell"
Technical Report SETC/IN/213
STL NorthWest, Kidsgrove

Budde R, Kuhlenkamp K, Mathiassen L, Zullighoven H - editors (1984)
Approaches to Prototyping
publ. Springer-Verlag, Berlin

Burns A, Robinson J (1986)
"ADDS - a dialogue development system for the Ada programming language"
International Journal of Man-Machine Studies 24, 2 pp.153-170

Bury K (1984)
"The iterative development of usable computer interfaces"
in [INTERACT 84] pp.743-750

Buxton W, Lamb M R, Sherman D, Smith K C (1983)
"Towards a comprehensive user interface management system"
ACM Computer Graphics 17, 3 pp.35-42

Carbonnell J R (1970)
"AI in CAI: an artificial intelligence approach to computer-aided instruction"
IEEE Transactions on Man-Machine Systems MMS-11, 4 pp.190-202

Card S K, Moran T P, Newell A (1980)
"The keystroke-level model for user performance time with interactive systems"
Communications of the ACM 23, 7 pp.396-410

Card S K, Moran T P, Newell A (1983)
The Psychology of Human-Computer Interaction
publ. Lawrence Erlbaum Associates, New Jersey

Cardelli L, Pike R (1985)
"Squeak - a language for communicating with mice"
ACM Computer Graphics 19, 3 pp.199-204

Carey T T, Mason R E A (1983)
"Information systems prototyping: techniques, tools, and methodologies"
INFOR 21, 3 pp.177-191

Carey T (1984)
"Dialogue handling with user workstations"
in [INTERACT 84] pp.127-134

Casey B E, Dasarathy B (1982)
"Modelling and validating the man-machine interface"
Software - Practice and Experience 12, 6 pp.557-569

CHI (1983)
Proceedings Conference on Human Factors in Computer Systems (CHI'83)
ed. A.Janda; publ. North-Holland, Amsterdam

CHI (1985)
Proceedings Conference on Human Factors in Computer Systems II (CHI'85)
ed. L.Borman, W.Curtis; publ. North-Holland, Amsterdam

CHI (1986)
Proceedings Conference on Human Factors in Computer Systems III
(CHI'86)
publ. ACM, New York

Chi U H (1985)
"Formal specification of user interfaces: a comparison and evaluation of four axiomatic approaches"
IEEE Transactions on Software Engineering 11, 8 pp.671-688

Christensen N, Kreplin K (1984)
"Prototyping of user interfaces"
in [Budde et al 84] pp.59-67

Clark R G (1986)
"Ada programs from me too specifications"
Technical Report TR.30
Department of Computing Science, University of Stirling

Cockton G (1986)
"Where do we draw the line?"
in [HCI 86] pp.417-432

Cohen D, Swartout W, Balzer R (1982)
"Using symbolic execution to characterize behaviour"
in [Squires 82] pp.25-32

Cook S (1986)
"Modelling generic user-interfaces with functional programs"
in [HCI 86] pp.369-385

Cox B J (1986)
Object-oriented programming
publ. Addison-Welsey, Massachusetts

Damodaran L, Eason K D (1983)
"Procedures for user involvement and support"
in [Sime & Coombs 83] pp.373-388

Darlington J (1981)
"An experimental program transformation and synthesis"
Artificial Intelligence 16, 16, 1 pp.1-46

Darlington J, Henderson P, Turner D A - editors (1982)
Functional Programming and its Applications: an advanced course
publ. Cambridge University Press, Cambridge

Darlington J (1985)
"Program transformation"
Byte 10, 8 pp.201-214

Davis G B (1982)
"Strategies for information requirements determination"
IBM Systems Journal 21, 1 pp.4-30

Dearnley P A, Mayhew P J (1983)
"In favour of system prototypes and their integration into the systems development cycle"
The Computer Journal 26, 1 pp.36-42

Degano P, Sandewall E - editors (1983)
Integrated Interactive Computing Systems
publ. North-Holland, Amsterdam

Denert E (1977)
"Specification and design of dialogue systems with state diagrams"
in Proceedings International Computing Symposium 1977 pp.417-424
ed. D.Ribbens
publ. North-Holland, Amsterdam

Denvir B T, Harwood W T, Jackson M I, Wray M J - editors (1985)
The Analysis of Concurrent Systems (LNCS 207)
publ. Springer-Verlag, Berlin

Dijkstra E W (1975)
"Guarded commands, non-determinacy and formal derivation of programs"
Communications of the ACM 18, 8 pp.453-457

Dix A, Runciman C (1985)
"Abstract models of interactive systems"
in [HCI 85] pp.13-22

Duce D A - editor (1984)
Distributed Computing Systems Programme
publ. Peter Peregrinus Ltd, London

Duce D A, Fielding E V C (1984)
"Better understanding through formal specification"
Technical Report RAL-84-128; Rutherford-Appleton Laboratory

Durham A (1985)
"User-shaped software"
Computing - the magazine July 25 pp.5-6

Durrett J, Stimmel (1982)
"A production-system model of human-computer interaction"
in [Gaithersburg 82] pp.393-399

Edmonds E A (1982)
"The man-computer interface: a note on concepts and design"
International Journal of Man-Machine Studies 16, 3 pp.231-236

Edmonds E A, Guest S (1984)
"The SYNICS2 user interface manager"
in [INTERACT 84] pp.53-56

Ehrig H, Floyd C, Nivat M, Thatcher J - editors (1985)
Formal Methods and Software Development
(LNCS 186; Proceedings TAPSOFT Conference, vol.2)
publ. Springer-Verlag, Berlin

Feather M S (1982)
"Mappings for rapid prototyping"
in [Squires 82] pp.17-24

Feldman G (1982)
"Functional specifications of a text editor"
in Proceedings ACM Conference on Lisp and Functional Programming Languages pp.37-46

Feldman M B, Rogers G T (1982)
"Towards the design and development of style-independent interactive systems"
in [Gaithersburgh 82] pp.111-116

Feyock S (1977)
"Transition diagram-based CAI/HELP systems"
International Journal of Man-Machine Studies 9, 4 pp.339-413

Finn S (1984)
"The CONVRULES production rule compiler - a user manual"
Internal note
Dept. of Computing Science, University of Stirling

Floyd J D, van Dam A (1982)
Fundamentals of Interactive Computer Graphics
publ. Addison-Wesley, Massachusetts

de Francesco N, Latella D, Vaglini G (1985)
"An interactive debugger for a concurrent language"
in Proceedings 8th Int. Conference on Software Engineering pp.320-325
publ. IEEE

Frohlich D M, Crossfield L P, Gilbert G N (1985)
"Requirements for an intelligent form-filling interface"
in [HCI 85] pp.102-116

Gaines B R, Shaw M L G (1984)
The Art of Computer Conversation
publ. Prentice-Hall International, New Jersey

Gaines B R, Shaw M L G (1986b)
"Foundations of dialog engineering - the development of hci. (Part II)"
International Journal of Man-Machine Studies 24, 2 pp.101-123

Gaithersburg (1982)
Proceedings Conference on Human Factors in Computer Systems
Gaithersburg, Maryland, USA
publ. ACM, New York

Galitz W O (1985)
A handbook of screen format design
publ. QED, Massachusetts

Gehani N H (1983)
"High-level form definition in office information systems"
The Computer Journal 26, 1 pp.52-59

Gilb T, Weinberg G M (1977)
Humanized Input
publ. Winthrop

Goguen J A, Tardo J J (1979)
"An introduction to OBJ: a language for writing and testing formal algebraic program specifications"
in Proceedings Specification of Reliable Software pp.170-189

Goguen J A, Meseguer J (1982)
"Rapid prototyping in the OBJ executable specification language"
in [Squires 82] pp.75-84

Goguen J A (1984)
"Parameterized programming"
IEEE Transactions on Software Engineering SE-10, 5 pp.528-543

Goldberg A, Robson D (1983)
Smalltalk-80: the language and its implementation
publ. Addison-Wesley, Massachusetts

Goltz U, Reisig W (1984)
"CSP-programs as nets with individual tokens"
in Advances in Petri Nets (LNCS 188) pp.169-196
ed. G.Rozenberg; publ. Springer-Verlag, Berlin

Gomaa H, Scott D B (1981)
"Prototyping as a tool in the specification of user requirements"
in Proceedings 5th IEEE Int. Conference on Software Engineering
pp.333-342
publ. IEEE, New York

Gomaa H (1983)
"The impact of rapid prototyping on specifying user requirements"
ACM SIGSOFT Software Engineering Notes 8, 2 pp.17-28

Good M (1981)
"Etude and the folklore of user interface design"
ACM SIGPLAN Notices 16, 6 pp.34-43

Good M, Whiteside J, Wixon S, Jones S (1984)
"Building a user-derived interface"
Communications of the ACM 27, 10 pp.1032-1043

Gorski J (1985)
"A technique for formal specification of parallel systems based on message-passing semantics"
in 3rd Int. Workshop on Software Specification & Design pp.77-82

Gray P M D (1984)
Logic, Algebra and Databases
publ. Ellis Horwood Limited, Chichester

Gray P, Kilgour A (1985)
"GUIDE - a UNIX-based dialogue design system"
in [HCI 85] pp.148-160

Green M (1985)
"The design of graphical user interfaces"
Technical Report CSRI-170 (PhD Thesis)
Computer Science Research Institute
University of Toronto, Canada

Gregory S T (1984)
"On prototypes vs. mockups"
ACM, SIGSOFT Software Engineering Notes 9, 5 p.13

Guedj R A, ten Hagen P J W, Hopgood F R A, Tucker H A, Duce D A - editors (1980)
Methodology of Interaction
publ. North-Holland, Amsterdam

Guest S P (1982)
"The use of software tools for dialogue design"
International Journal of Man-Machine Studies 16, 3 pp.263-285

Guttag J V, Horning J J (1978)
"The algebraic specification of abstract data types"
Acta Informatica 10, 1 pp.27-52

Guttag J V, Horning J J (1980)
"Formal specification as a design tool"
in Proceedings Symposium on Principles of Programming Languages
pp.251-261; publ. ACM, New York

Guttag J V, Horning J J, Wing J (1982)
"Some notes on putting formal specifications to productive use"
Science of Computer Programming 2, 1 pp53-58

Haase V H (1985)
"Modular design of real-time systems"
in System Description Methodologies pp.91-102
ed. D.Teichroew, G.David

ten Hagen P J W, Derksen J (1985)
"Parallel input and feedback in dialogue cells"
in [Pfaff 85] pp.109-124

Hagglund S, Tibell R (1983)
"Multi-style dialogues and control independence in interactive software"
in The Psychology of Computer Use pp.171-189
ed. T R G Green et al

Hammond N, Jorgensen A, MacLean A, Barnard P, Long J (1983)
"Design practice and interface usability: evidence from interviews with designers"
in [CHI 83] pp.40-44

Hanau P R, Lenorovitz D R (1980)
"Prototyping and simulation tools for user/computer dialogue description"
ACM Computer Graphics 14, 3 pp.271-278

Harel D (1986)
"Statecharts: a visual approach to complex systems"
Technical Report CS86-02
Dept. of Applied Mathematics, Weizmann Institute of Science, Israel

Hartson H R, Johnson D H, Ehrich R W (1984)
"A human-computer dialogue management system"
in [INTERACT 84] pp.379-383

Hartson H R - editor (1985)
Advances in Human-Computer Interaction (vol.1)
publ. Ablex Publishing Corp., New Jersey

Hayes P J (1985)
"Executable interface definitions using form-based interface abstractions"
in [Hartson 85] pp.161-190

HCI (1985)
People and Computers: Designing the Interface, (Proceedings HCI'85)
ed. P.Johnson, S.Cook
publ. Cambridge University Press, Cambridge

HCI (1986)
People and Computers: Designing for Usability, (Proceedings HCI'86)
ed. M.Harrison, A.F.Monk
publ. Cambridge University Press, Cambridge

Hekmatpour S, Ince D (1986a)
"A formal specification-based prototyping system"
in Software Engineering'86 (Proceedings) pp.317-335
ed. D.Barnes, P.Brown
publ. Peter Peregrinus Ltd, London

Hekmatpour S, Ince D (1986b)
"Rapid software prototyping"
Technical Report 86/4
Computer Discipline, Faculty of Maths
Open University, Milton Keynes

Henderson P (1980)
Functional Programming: application and implementation
publ. Prentice-Hall International, New Jersey

Henderson P (1982)
"Purely functional operating systems"
in [Darlington et al 82] pp.177-192

Henderson P (1984)
"Communicating functional programs"
Internal Report FPN-8
Dept. of Computing Science, University of Stirling

Henderson P, Minkowitz C (1986)
"The me too method of software design"
ICL Technical Journal May pp.64-95

Henderson P, Minkowitz C J, Rowles J S (1985)
me too Reference Manual
SETC, ICL Kidsgrove

Ho T-P (1984)
"The dialogue designing dialogue system"
PhD Thesis, California Institute of Technology

Hoare C A R (1978)
"Communicating sequential processes"
Communications of the ACM 21, 8 pp.666-677

Hoare C A R (1982a)
"Programming is an engineering profession"
Technical Monograph PRG-27
Oxford University Programming Research Group
University of Oxford

Hoare C A R (1982b)
"Specifications, programs and implementations"
Technical Monograph PRG-29
Oxford University Programming Research Group
University of Oxford

Hoare C A R (1983)
"Notes on Communicating Sequential Processes"
Technical Monograph PRG-33
Oxford University Programming Research Group
University of Oxford

Hoare C A R (1985)
Communicating Sequential Processes
publ. Prentice-Hall International, New Jersey

Hoare C A R, Shepherdson J C - editors (1985)
Mathematical Logic and Programming Languages
publ. Prentice-Hall International, New Jersey

Hopgood F R A, Duce D A (1980)
"A production system approach to interactive graphic program design"
in [Guedj et al 80] pp.247-263

Horning J J (1985)
"Combining algebraic and predicative specifications in Larch"
in [Ehrig et al 85] pp.12-26

Huckle B A, Bull G M (1984)
"A model for software descriptions facilitating man-machine interface variations"
ACM SIGCHI Bulletin 16, 2 pp.70-75

Hull M E, McKeag R M (1984)
"Communicating sequential processes for centralised and distributed operating system design"
ACM Transactions on Programming Languages and Systems 6, 2 pp.175-191

ICL (1986)
"ICL application development"
Sales information; ICL, London

INMOS (1984)
occam Programming Manual
publ. Prentice-Hall International, New Jersey

INTERACT (1984)
Human-Computer Interaction (Proceedings INTERACT'84)
ed. B.Shackel
publ. Elsevier (North-Holland), Amsterdam

Iverson K E (1979)
"Notation as a tool of thought"
Communications of the ACM 23, 8 pp.445-465

Jacob R J K (1983)
"Survey and examples of specification techniques for user-computer interfaces"
Draft Report
Naval Research Laboratory, Washington DC

Jacob R J K (1985)
"An executable specification technology for describing human-computer interaction"
in [Hartson 85] pp.211-242

Jacob R J K (1986)
"A specification language for direct manipulation user interfaces"
Draft Report
Naval Research Laboratory, Washington DC

Jensen R W, Tonies C C - editors (1979)
Software Engineering
publ. Prentice-Hall International, New Jersey

Johnson D H, Hartson H R (1982)
"The role and tools of a dialogue author in creating human-computer interfaces'
Technical Report CSIE-82-8
Department of Computer Science
Virginia Polytechnic Institute and State University

Johnson D H (1985)
"The structure and development of human-computer interfaces"
PhD thesis, Virginia Polytechnic Institute & State University

Johnson S C (1978)
"YACC - Yet Another Compiler-Compiler"
Unix Programmers' Manual (Vol.2)

Johnson S C, Lesk M E (1978)
"Language development tools"
Bell Systems Technical Journal 57, 6 pp.2155-2175

Jones C B (1980)
Software Development - a rigorous approach
publ. Prentice-Hall International, New Jersey

Jones C B (1986)
Systematic Software Development using VDM
publ. Prentice-Hall International, New Jersey

Jones S B (1984)
"A range of operating systems written in a purely functional style"
Technical Report TR.16
Dept. of Computing Science, University of Stirling

Jones V M, Jones S B, Minkowitz C J (1985)
"A formal specification of an expert system shell"
Technical Report TR.20
Dept. of Computing Science, University of Stirling

Kamran A (1985)
"Issues pertaining to the design of a UIMS"
in [Pfaff 85] pp.43-48

Kasik D J (1982)
"A user interface management system"
ACM Computer Graphics 16, 3 pp.99-106

Kelley J F (1985)
"Validating an empirical methodology for writing user-friendly natural language computer applications"
IBM Research Report RC-10127 (45001)

Kemmerer R A (1985)
"Testing formal specifications to detect design errors"
IEEE Transactions on Software Engineering SE-11, 1 pp.34-42

Kieras D, Polson P G (1985)
"An approach to the formal analysis of user complexity"
International Journal of Man-Machine Studies 22, 4 pp.365-394

Kowalski R (1985)
"The relation between logic programming and logic specification"
in [Hoare & Shepherdson 85] pp.11-24

Lafuente J M, Gries D (1978)
"Language facilities for programming user-computer dialogues"
IBM Journal of Research and Development 22, 2 pp.145-158

Lawson H W, Bertran M, Sanagustin J (1978)
"The formal definition of human/machine communications"
Software - Practice and Experience 8, 1 pp.51-58

Lee S, Sluizer S (1985)
"On using executable specifications for high-level prototyping"
in 3rd Int. Workshop on Software Specification & Design pp.130-134

Lieberman H (1983)
"Designing interactive systems from the user's viewpoint"
in [Degano & Sandewall 83] pp.45-59

Lieberman H (1985)
"There's more to menu systems than meets the screen"
in ACM Computer Graphics 19, 3 pp.181-189

Lindquist T (1985)
"Assessing the usability of human-computer interfaces"
IEEE Software 2, 1 pp.74-82

Liskov B, Zilles S (1975)
"Specification techniques for data abstractions"
ACM SIGPLAN Notices 10, 6 pp.72-87

Mallgren W R (1983)
Formal Specification of Interactive Graphics Programming Languages
publ. MIT Press, Massachusetts

Manna Z, Pneuli A (1981)
"Verification of concurrent programs: the temporal framework"
in The Correctness Problem in Computer Science pp.215-273
ed. R.S.Boyer, J.S.Moore
publ. Academic Press, London

Martin J (1973)
Design of Man-Computer Dialogues
publ. Prentice-Hall International, New Jersey

Mason R E, Carey T T (1983)
"Prototyping interactive information systems"
Communications of the ACM 26, 5 pp.347-354

McCarthy J (1960)
"Recursive functions of symbolic expressions and their computation by machine"
Communications of the ACM 3, 4 pp.184-195

Meadow C T (1970)
Man-Machine Communication
publ. John Wiley & Sons

Milner (1985)
"Using algebra for concurrency : some approaches"
in [Denvir et al 85] pp.7-25

Minkowitz C J (1986)
"A formal design of a decision analysis system"
Technical Report TR.27
Dept. of Computing Science, University of Stirling

Minkowitz C J, Henderson P (1986)
"A formal description of object-oriented programming using VDM"
Internal Report FPN-13
Dept. of Computing Science, University of Stirling

Minsky M L (1967)
Computation : finite and infinite machines
publ. Prentice-Hall International, New Jersey

Minsky M L (1975)
"A framework for representing knowledge"
in The Psychology of Computer Vision pp.211-277
ed. P.H. Winston, publ. McGraw-Hill, New York

Moran T P (1981a)
"The Command Language Grammar - a representation for the user interface of interactive computer systems"
International Journal of Man-Machine Studies 15, 1 pp.3-50

Moran T P (1981b)
"An applied psychology of the user"
ACM Computer Surveys 13, 1 pp.1-12

Moskowski N (1986)
Executing temporal logic specifications
publ. Cambridge University Press, Cambridge

muLisp (1983)
muLisp 83 Reference Manual
publ. Microsoft Corp.

Naur P (1982)
"Formalization in program development"
BIT 22, 4 pp.437-453

Neal A S, Simons R M (1983)
"Playback: a method for evaluating the usability of software and its documentation"
in [CHI 83] pp.78-82

Neely R (1983)
"A protocol simulation tool"
MSc thesis, Oxford University

Nielsen J (1986)
"A virtual protocol model for computer-human interaction"
International Journal of Man-Machine Studies 24, 3 pp.301-312

Norman D A (1984)
"Cognitive engineering principles in the design of human-computer interfaces"
in [Salvendy 84] pp.11-16

Olsen D R, Dempsey E P (1983)
"Syntax directed graphical interaction"
ACM SIGPLAN Notices 18, 6 pp.112-117

Olsen D R (1984)
"Pushdown automata for user interface management"
ACM Transactions on Graphics 3, 3 pp.177-203

Olsen D R, Dempsey E P, Rogge R (1985)
"Input/output linkage in a User Interface Management System"
ACM Computer Graphics 19, 3 pp.191-197

Orr W D - editor (1968)
Conversational Computers
publ. John Wiley & Sons

Otte F (1982)
"Consistent user interface"
in [Vassilou 82] pp.261-276

Parnas D L (1969)
"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
publ. ACM, New York

Patton B (1983)
"Prototyping - a nomenclature problem"
ACM SIGSOFT Software Engineering Notes 8, 2 pp.14-16

Payne S J, Green T R G (1983)
"The user's perception of the interaction language: a two-level model"
in [CHI 83] pp.202-206

Pfaff G E - editor (1985)
User Interface Management Systems
publ. Springer-Verlag, Berlin

Pressman R S (1982)
Software Engineering: a practitioner's approach
publ. McGraw-Hill

Reid P (1985)
Trip report on CHI'85 conference
Alvey Software Engineering mailshot, August 1985

Reisner P (1983)
"Formal grammar as a tool for analysing ease of use: some fundamental concepts"
in Human Factors in Computer Systems pp.53-78
ed. J.C.Thomas, M.Schneider
publ. Ablex, New Jersey

Rowles J S (1986)
"Describing screen layouts in a purely functional style"
Technical Report SETC/IN/217
STL Northwest, Kidsgrove

Salvendy G - editor (1984)
Human-Computer Interaction
(Proceedings 1st USA-Japan Conference)
publ. Elsevier, Amsterdam

Sandewall E (1982)
"Unified dialogue management in the Carousel system"
in Office Information Systems pp.175-197
ed. N.Naffah
publ. North-Holland, Amsterdam

Schneider M (1982)
"Ergonomic considerations in the design of command languages"
in [Vassilou 82] pp.141-161

Shackel B (1986)
"IBM makes usability as important as functionality"
The Computer Journal 29, 5 pp.475-476

Shaw A C (1980)
"On the specification of graphics command languages and their processors"
in [Guedj et al 80] pp.377-392

Shaw M, Borison E, Horowitz M, Lane T, Nichols D, Pausch R (1983)
"Descartes - a programming language approach to interactive display interfaces"
ACM SIGPLAN Notices 18, 6 pp.100-111

Sheeran M (1984)
"muFP, an language for VLSI design"
in Proceedings Conference on Lisp & Functional Languages pp.104-112

Shneiderman B (1982a)
"The future of interactive systems and the emergence of direct manipulation"
in [Vassilou 82] pp.1-27

Shneiderman B (1982b)
"Multiparty grammars and related features for defining interactive systems"
IEEE Transactions on Systems, Man and Cybernetics SMC-12, 2 pp.148-154

Sime M E, Coombs M J (1983)
Designing for Human-Computer Communication
publ. Academic Press, London

Smith D A (1982)
"Rapid software prototyping"
PhD thesis, Univesity of California, Irvine

Smith R G, Lafue G M E, Schoen E, Vestal S C (1984)
"Declarative task description as a user-interface structuring mechanism"
Computer 17, 9 pp-29-38

Sommerville I (1982)
Software Engineering
publ. Addison-Wesley, London

Sproull R F (1983)
"Programming the user interface"
in Proceedings Joint IBM/University of Newcastle-Upon-Tyne Seminar pp.135-143
University of Newcastle-Upon-Tyne

Squires S L - editor (1982)
ACM SIGSOFT Software Engineering Notes 7, 5
(Special issue on Rapid Prototyping)

Stavely A M (1982)
"Models as executable designs"
in [Squires 82] pp.167-168

Strand E M, Jones W T (1982)
"Prototyping and small-scale projects"
in [Squires 82] pp.169-170

Strom R, Yemini S (1985)
"The NIL distributed systems programming language: a status report"
ACM SIGPLAN Notices 20, 5 pp.36-44

Strubbe H J (1985)
"Components of interactive applications"
in [Pfaff 86] pp.49-57

Studer R (1984)
"Abstract models of dialogue concepts"
in Proceedings 7th IEEE Int. Conference on Software Engineering
pp.420-429
publ. IEEE

Sufrin B (1982)
"Formal specification of a display-oriented text editor"
Science of Computer Programming 1, 3 pp.157-202

Sutton J A, Sprague R H (1978)
"A study of display generation and management in interactive business applications"
IBM Research Report RJ2392

Swartout W, Balzer R (1982)
"On the inevitable intertwining of specification and implementation"
Communications of the ACM 25, 7 pp.438-440

Tavendale R D (1985)
"A technique for prototyping directly from a specification"
in Proceedings 8th Int. Conference of Software Engineering
pp.224-229; publ. IEEE

Taylor T, Standish T A (1982)
"Initial thoughts on rapid prototyping techniques"
in [Squires 82] pp.160-166

Thiagarajan P S (1985)
"Some aspects of net theory"
in [Denvir et al 85] pp.26-54

Thimbley H (1982)
"Dialogue determination"
International Journal of Man-Machine Studies 13, 3 pp.295-304

Thomas J C (1982)
"Organising for human factors"
in [Vassilou 82] pp.29-46

Turner D A (1982)
"Recursion equations as a programming language"
in [Darlington et al 82] pp.1-28

Turner D A (1985)
"Functional programs as executable specifications"
in [Hoare & Shepherdson 85] pp.29-54

Underwood M (1985)
"Alvey MMI - opportunities for multi-disciplinary research"
Seminar, University of Glasgow

Vassilou Y - editor
Human Factors in Interactive Computer Systems
publ. Ablex, New Jersey

Wadge W W, Ashcroft E A (1985)
Lucid, the dataflow programming language
publ. Academic Press, London

Wartik S P, Pyster A (1983)
"The 'diversion' concept in interactive computer systems specifications"
in Proceedings IEEE 7th Int.Computer Software and Applications Conference (COMPSAC '83) pp.281-286; publ. IEEE

Wasserman A I, Shewmake D T (1982)
"Rapid prototyping of interactive information systems"
in [Squires 82] pp.171-180

Wasserman A I (1985)
"Extending state transition diagrams for the specification of human-computer interaction"
IEEE Transactions on Software Engineering SE-11, 8 pp.699-713

Wasserman A I, Pircher P A, Shewmake D T, Kersten M L (1986)
"Developing interactive information systems with the User Software Engineering methodology"
IEEE Transactions on Software Engineering SE-12, 2 pp.326-345

Weiser M (1982)
"Scale models and rapid prototyping"
in [Squires 82] pp.181-185

While L (1986)
"Synchronisation in functional languages"
Draft report, Imperial College, London

Williges R C (1984)
"Design of human-computer dialogues"
in [Salvendy 84] pp.35-42

Woods W A (1970)
"Transition network grammars for natural language"
Communications of the ACM 13, 10 pp.591-606

Yunten T, Hartson H R (1985)
"A SUPERvisory Methodology And Notation (SUPERMAN) for human-computer system development"
in [Hartson 85] pp.243-282

Zave P, Schell W (1986)
"Salient features of an executable specification language and its environment"
IEEE Transactions on Software Engineering SE-12, 2 pp.312-325