The Development of an Executable Graphical Notation for Describing Direct Manipulation Interfaces

A dissertation presented
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy in Computer Science
at Massey University

Christopher Henry Edwin Phillips

1993

Abstract

The research reported in this thesis involves the development of an executable semi-formal graphical notation, Lean Cuisine+, for describing the underlying behaviour of event-based direct manipulation interfaces, and the application of the notation both in 'reverse engineering', and during the early design phase of the interface development life cycle.

A motivation for the research stems from the need for tools and techniques to support high level interface design. The research supports and brings together a number of views concerning the requirements of notations at this level. These are that a notation should be semi-formal, graphical, executable, and object-oriented, and that to be most effective it should be targeted at a specific category of interaction. The Lean Cuisine+ notation meets all these criteria, the underlying meneme model matching closely with the selection-based nature of direct manipulation interfaces.

Lean Cuisine+ is a multi-layered notation, and is a development of Lean Cuisine (Apperley & Spence, 1989). The base layer is a tree diagram which captures part of the behaviour of an interface in terms of constraints and dependencies between selectable dialogue primitives. Further constraints and dependencies associated with the dynamics of the interface are captured through overlays to the basic tree diagram. An orthogonal task layer captures any temporal relationships between primitive task actions, and provides a link with higher level functionality. Lean Cuisine+ is able to combine both static and dynamic modelling in a coherent manner, thus avoiding the necessity of employing separate and possibly disjoint models at the early design stage. A software support environment for the notation is also specified and partially prototyped.

The research demonstrates the advantages of a notation which can be executed to provide limited but valid early simulation of the dynamic behaviour of the interface under design. A mapping from Lean Cuisine+ to a dialogue implementation language, DAL (Anderson, 1993), is also developed in support of the view that a multi-notational approach to interface development is required, and that it must be possible to move easily from initial specification to prototyping and implementation.

The Lean Cuisine+ descriptions of aspects of the Apple Macintosh interface included in the thesis show the notation to be capable of handling a range of direct manipulation interaction styles and a variety of interface objects. A five stage methodology for the construction of Lean Cuisine+ specifications for new interfaces is also developed, and applied in two case studies.

Acknowledgements

I wish to thank Professor Mark Apperley, not only as my chief supervisor for his guidance, support and enthusiasm throughout this research, but also in his capacity as head of the Computer Science Department for creating the space which permitted its completion within a manageable time frame.

Thanks must also go to my second supervisor Dr Peter Kay for his painstaking reading and correction of the thesis, and to Philip Etheridge for the occasions on which he was called upon, often at short notice, to assist with the interfacing of Macintosh software. The assistance of Paul Anderson with regard to the mapping of Lean Cuisine+ into his Dialogue Activation Language is also appreciated.

The receipt of a grant from the Massey University Research Fund for the purchase of an A4 portrait monitor to support aspects of this research is also gratefully acknowledged.

On a more personal level I am grateful for the patience and support of my wife Carol throughout the research, and to my daughters for the many hours at home when because of the research I was out of circulation.

Finally, I wish to dedicate this thesis to my parents, Harry and Dorothy, for their vision in wanting me to have an education, and for supporting me through school and university at considerable personal cost.

Brief Contents

Detailed Contents

Figures and Tables

Figures

Tables

Part 1

Survey and Framework

Chapter 1

Introduction

"The problem addressed here is not how to construct good interfaces; it is how to provide an environment in which good interfaces can be constructed."

Hartson & Hix, 1989

1.1 Motivation for the research

As human-computer interfaces have become easier to use they have become harder to create. Graphical interface software is large, complex, and difficult to debug and modify. The design of graphical interfaces has been likened to the design of buildings - partly an art, partly a science (Foley & Van Dam, 1984). An application's interface can account for a significant fraction of the code. Surveys of artificial intelligence applications, for example, report that 40 or 50 percent of the code and run-time memory are devoted to interface aspects (Bobrow, Mittal & Stefik, 1986). Smith (1986) points out that user interface software is not only critical to system performance but can also represent a sizeable investment. The production of good interactive software is therefore both difficult and expensive.

Rhyne & Watson (1987) consider that interface requirements are rarely as well defined and stable as requirements for computational algorithms. Farooq & Dominick (1988, p.481) are also of the view that the creation of a user interface requires special skills, tools and methodologies, "because it is an intrinsically different activity from the coding of computational algorithms". In the context of menu systems, Edmondson & Spence (1992, p.1) observe that "a designer has available, on the one hand, an embarrassingly large repertoire of interaction and presentation techniques, but on the other, a distressingly small toolkit of guidelines, notations, methodologies, frameworks and languages to provide the means for intelligent thought".

Hartson & Hix (1989, p.8) define dialogue "as the observable two-way exchange of symbols and actions between human and computer". In this thesis the term dialogue is used to refer to the *structure* of the human-computer interaction expressed in terms of constraints relating to dialogue primitives, and described through the employment of a formal or semi-formal dialogue model. Structural models of the human-computer interface serve as frameworks for understanding the elements of interfaces and for

guiding the dialogue developer in their construction (Hartson, 1989). Rhyne & Watson (1987) take the view that some symbolic specification of interface behaviour appears inescapable for the near future, but that the task is difficult, and can only be made easier through the development of better specification paradigms. A problem identified with some existing dialogue notations is that they describe only the 'surface behaviour' of the interface. New notations must go beyond this and must be capable of associating the functionality of the interface with objects, actions and states. Two concepts associated with dialogue design underpin the research reported. The first is the notion of 'dialogue independence' - the separation of the design of dialogue from the design of computational software. Most current approaches to interface development are based to some extent on dialogue independence. The second is the isolation of the *design* of interactive dialogues from the detail of their implementation.

Interface representation is currently achieved through a variety of notational schemes, including both textual and graphical representation languages. Control structures govern how sequencing among dialogue and computational components is designed and executed, the two basic kinds being 'sequential' and 'asynchronous' (Hartson, 1989). In sequential dialogue only one task is presented to the user at any one time, whereas in asynchronous dialogue more than one task or thread may be available. Asynchronous dialogue is also described as multi-threaded. In this thesis it is assumed to be multi-threaded dialogue *in which there is generally an absence of sequence within the separate threads or subdialogues*. Asynchronous dialogue is also referred to as 'event-based' dialogue (Hartson & Hix, 1989, p.11), "because end-user actions (e.g., clicking the mouse button on an icon) are viewed as input events". It should be noted that asynchronous does not imply concurrent. Concurrent dialogue is multi-threaded dialogue in which more than one thread can be executed simultaneously.

Direct manipulation interfaces

So-called 'direct manipulation' interfaces are asynchronous, with the objects of interest being displayed so that actions are directly in the problem domain. They are most easily understood by example, perhaps the best known being the Apple Macintosh interface (Apple, 1989a) which is used as the basis for the case studies in this thesis. This interface is based on the 'desktop' metaphor in which icons represent documents and folders and in which the mouse is the electronic extension of the hand. This is an example of the more general 'model world' metaphor defined by Hudson & King (1986). Many real direct manipulation interfaces include elements from several styles in a single interface, e.g. menus, dialogue boxes and windows; incorporate objects of a dynamic nature, e.g. windows and icons; and involve concurrency, conditions, and object inter-relationships. The nature of direct manipulation interfaces is examined in detail in Chapter 2 of this thesis.

The specification of direct manipulation interfaces presents particular problems not encountered in other software systems. Myers (1989, p.15) states that "direct manipulation interfaces popular on many modern systems are among the most difficult to implement....because they often provide elaborate graphics, many ways to give the same command, many asynchronous input devices, a mode-free interface (the user can give any command at virtually any time), and rapid semantic feedback". Rhyne & Watson (1987) highlight the difficulties of testing for potential ambiguities in the design of asynchronous dialogues, and identify a need for tools to assist the designer in building and debugging such dialogues. Hartson & Hix (1989, p.18) see research on structural modelling of asynchronous dialogue as still embryonic, because "such dialogue is less structured than sequential dialogue". Techniques are required for recording behavioural, structural and detailed representation of both visible and non-visible aspects of direct manipulation interfaces. Ideally the techniques should be independent of tools through which they may be implemented, and be complete in their ability to represent interfaces.

The 'object-oriented' paradigm, in which hierarchies of objects are manipulated and where immediate semantic feedback is required, seems to suit the implementation of direct manipulation interfaces (Barth, 1986; Sibert, Hurley & Bleser, 1986; Rhyne, Ehrich, Bennett, Hewett, Sibert & Bleser, 1987; Hartson, 1989; Foley, Kim, Kovacevic & Murray, 1989; Bass & Coutaz, 1991; Markopoulos, Pycock, Wilson & Johnson, 1992). In describing the development of the Xerox Star interface, Smith, Irby, Kimball, Verplank & Harslem (1982) argue that such interfaces are by their very nature object-oriented. Hartson takes the view that object orientation is effective for representing asynchronous dialogue, and for representing the behaviour of specific interface features (e.g. windows), because of its event-based nature. However, he also considers that a disadvantage of object orientation is its tendency to obscure temporal relationships in the high level sequencing behaviour in the application interface.

Event-based mechanisms are currently the primary underlying techniques upon which asynchronous dialogue is constructed. Reliable guidelines for the *design* of direct manipulation interfaces do not yet exist. There is a need for tools that can assist with their design and development.

The interface development life cycle (IDLC)

The interface development process, or life cycle, can be viewed as consisting of a number of stages. Shneiderman (1987) defines an eight stage life cycle, consisting of: collection of information; definition of requirements and semantics; design of syntax and support facilities; specification of physical devices; development of software; implementation and testing; evaluation; and revision. The need to view interface development as an integral part of the software engineering process is being recognised, and this life cycle bears broad resemblance to the more generally prescribed software life

cycle (e.g. Sommerville, 1985). Although this is a step-by-step description, the process is equally iterative. A distinction must be drawn between design representations and design methodologies. The former define the space of objects that can be designed within them and impose structure on them (Moran, 1981). A representation may be incorporated into a methodology, and it is important to show where such representations fit into the scheme of things.

A variety of models at various levels of abstraction have been employed in the early stages of the IDLC. These include user models, task models, and dialogue models. From an implementation viewpoint, Green (1986) broadly classifies the latter into state transition networks, formal grammars, and event-based techniques. User and task models are applicable to the first two stages of the life cycle, that is, to information collection and requirements definition. They form important inputs to the early phase of Shneiderman's third (design) stage, where dialogue models come into play, and where prototyping of the interface may be employed. Prototyping is also sometimes called dialogue simulation, and prototypes may involve executable specifications. Hartson & Hix (1989) view prototyping as an effective way to begin evaluation and testing earlier in the life cycle than was formerly the case. Dialogue models and notations, which are the focus of this thesis, are reviewed in Chapter 3.

Existing interface development tools

Many attempts have been made to develop tools to make the task of designing and implementing interfaces easier. These include user interface toolkits, and so-called 'user interface management systems' (UIMSs). Toolkits are libraries of pre-defined interaction techniques offering support for the implementation of limited interaction styles, but providing only minimal support for interface design. UIMSs reach further back into the IDLC by providing for the specification of sequencing and dialogue control. In UIMSs, the concept of dialogue independence is explicitly recognised and supported. Both UIMSs and toolkits are intended to speed up the development of interfaces. Hill (1987) defines the key goals of a UIMS as reducing the high cost of implementing user interfaces, and helping improve the quality of user interfaces by facilitating prototyping and experimentation. Hix & Hartson (1986) consider however that many UIMSs emphasise the execution-time aspects of interface management at the expense of interface design. Koiuvnen & Mantyla (1988, p.44) state that "the concept of UIMS is still also a vague one, and many mutually inconsistent approaches exist". They go on to produce a taxonomy of toolkits and UIMSs. Myers (1989) also provides a useful survey of both toolkits and UIMSs.

All UIMSs are restricted in the forms of user interfaces they can generate. A further limitation of many UIMSs is that they require the designer to specify interfaces in a textual, formal, programming-style language (Myers, 1987b). This has proved useful

and appropriate for textual command languages but difficult and clumsy for direct manipulation interfaces. A number of UIMSs allow the designer to use more graphical styles. Examples include Menulay (Buxton, Lamb, Sherman & Smith, 1983), Trillium (Henderson, 1986), and GRINS (Olsen, Dempsey & Rogge, 1985). These are still limited for the most part to using graphical techniques for specifying static aspects of the interface, i.e. to control the *placement* of interaction objects.

A number of problems have been identified with current UIMSs and interface toolkits:

- they are generally aimed at the programmer rather than the designer (Olsen, 1987a, 1987b; Myers, 1987a, 1989; Rhyne *et al*, 1987; Cockton, 1987, 1990b);
- they have borrowed models from other areas of computer science (Cockton, 1987);
- they are difficult to use (Rhyne *et al*, 1987);
- they support only part of the IDLC (Myers, 1989);
- they cannot support the development of direct manipulation interfaces (Myers, 1987a, 1989);
- they are unavailable commercially or not portable (Myers, 1989);
- they do not support evaluation (Myers, 1989).

Cockton (1990a, 1990b) considers models and architectures to be central to the development of tool-based design environments for interactive systems, which he calls 'interactive system design environments' (ISDEs). Cockton believes that current software practice is still dominated by a 'programming language' approach, where the designer's representation of the formalism is the implementation language. Olsen (1987b, p.73) goes further, stating that "A UIMS can be viewed as a tool for increasing programmer productivity". Myers (1987a) even argues for a UIMS classification based on the level of programming skills needed to use them, which he sees as closely tied to their ease of use. Olsen (1987c) considers that difficulties in expressing interfaces in 'borrowed models' have led to UIMSs not being used, in spite of large potential productivity gains. Rhyne *et al* (1987) see the difficulties in using current UIMSs as arising partly from the difficulty of the problem, partly from the considerable programming skills required by users of today's tools, and partly because tool interfaces are themselves badly designed.

Most automated tools support the developer in the *coding* phase of software production, not the design phase, and they generally lack tools to facilitate construction of human-computer interfaces (Hartson, 1989). Markopoulos *et al* (1992) take the view that recent generations of UIMSs have provided the designer with increasingly sophisticated design tools supporting rapid prototyping. They go on to argue that prototyping is largely a trial and error software engineering activity to support implementation,

providing no explicit theory beyond the particular implementation, and lacking principles for design and design decisions. Models and tools must be developed that provide support for the design process, to enable a designer to evaluate the consistency and completeness of the definition. Rhyne *et al* (1987, p.83) argue that "consistency is more likely to result when designers have a clear and spare mental model of the interface".

It must be recognised that direct manipulation interfaces are intrinsically complex, and may include multiple states, simultaneous interaction events, complex constraints, communication with the application, and elaborate feedback. Hudson (1987) argues that syntax (for direct manipulation interfaces) should be in terms of individual objects, should be as simple as possible, and should involve physical actions such as pointing or dragging instead of more linguistic concepts. This implies that previous approaches based on state transition networks or grammars which control the overall interface are not good candidates for direct manipulation interfaces. Jacob (1986, p.287) supports this, stating that "it is unnatural, though possible, to describe the user interface of a direct manipulation system as a conventional dialogue by means of a syntax diagram or other such notation". Cockton (1987) considers that the real issue is find a language that captures the *user's* view of a direct manipulation interface as perspicuously as possible, and with as few ad hoc features and extensions to the specification technique as possible. He also considers (Cockton, 1990b) that models are of limited use when we do not really understand the properties which component abstractions should satisfy. The identification of these properties for direct manipulation interfaces is central to this research.

The need for 'high level' tools

It is not enough for the designer or user to 'patch together' an interface using a toolkit. In supporting this view, Rhyne *et al* (1987, p.85) argue that "The complexity of the design problem at this level precludes the simple assembly of interaction technique modules". There is a need for 'high level' tools, models, and techniques for the designer, providing for both interface creation and interface analysis. Foley (1987) defines high level user interface representation as embodying information about the interface in terms of objects, actions, relations, attributes, and pre- and post-conditions associated with the actions. Alexander (1987) considers it important that interface designers are able to experiment with different ideas at the early stages of the interface development process.

With one or two exceptions (e.g. the UofA UIMS (Singh & Green, 1991)), there are few tools to support high level design of user interfaces. To some extent this is a consequence of the variety of individual methodologies in use, but Rhyne *et al* (1987) consider the greater cause to be the fact that few researchers have attempted to build such tools. Both Cockton (1990b) and Guindon (1990) also support this view. Guindon

observes that high level design has seldom been empirically studied, and that it is poorly supported by software tools and environments available today. Farooq & Dominick (1988, p.491) argue that higher level directly executable specification techniques are needed to avoid burying the interface designer in low level details, "especially during the experimentation associated with prototyping such interfaces". Olsen (1987a) identifies a goal for future research as being the development of designer oriented tools - in particular, tools that automate the design process, that involve high level representations of the user interface, and that allow designers to produce different styles of user interfaces. The high level design of interfaces is arguably the weakest link in the chain, and is an under-researched area.

Parallels can be drawn with two other areas of systems design:

- *The development of data models for database design*
  In the 1960s and early 1970s databases were designed using various implementation models, (hierarchical, network, or relational). Not until the mid 1970s was attention devoted to the development of independent conceptual data models, e.g. the E-R model (Chen, 1976), for use in the early stages of data base design.
- *The production of systems requirements specifications*
  Tse & Pong (1991) argue that a requirements specification should be independent of the design and implementation of the target system, and that the supporting language should be behaviour-oriented and non-procedural. It should provide a means of improving the conceptual clarity of the problems, should allow us to model logical and physical characteristics separately, and should be expressible in a precise notation with a unique interpretation. They also present a case for a hierarchical framework to permit downwards movement from higher levels of abstraction to more detailed descriptions.

Olsen (1987b) takes the view that the use of high level specifications should simplify the production of user interfaces. However, referring to existing models created for specifying interactive tasks, he states (Olsen, 1987c) that there are very few guidelines or principles to direct high level design of interfaces. He sees most current UIMS developments as being driven not by the task model but rather by syntactic and device oriented models. Smith (1986) argues the importance of distinguishing the underlying logic or architectural form of user interface functions from the detailed presentation of those functions to the user. Anderson & Apperley (1991, p.2) state that "initially what is needed is some form of abstraction that permits the basic dialogue structure of the interface to be defined, and then the details....to be added as required". In the context of menu systems, Apperley & Spence (1989) identify the need for a design methodology for the dialogue designer which should in itself provide a good interface to

the designer, and which should reflect the *structure* of the dialogue rather than the underlying program.

In order to characterise at a high level the interaction between a user and a system an explicit representation of the *behaviour* of the system itself is needed (Kieras & Polson, 1983). Edmondson (1991, p.8) notes that this is *underlying* behaviour "as distinct from the *superficial* behaviour we observe when looking at the mechanics of the interface". A representation of the behaviour of an interactive system should be independent of the implementation, and should reflect structural properties such as hierarchy, possible modes, and consistent patterns of interaction, in addition to being easy to define and understand. This representation will involve both semantic and syntactic aspects of the interface. Olsen (1987b, p.75) takes the view that the semantics of an application are expressed in "the behaviour of the objects and actions of the user interface and in the relationship between these objects and actions". Chakravarty & Kleyn (1990) observe that the interference of events internal to a task with the events in a higher level task making use of it, or with tasks that are interleaved in a multi-threaded execution, is a major problem associated with developing a specification of high level system behaviour.

Formal vs informal specifications

There is a tension between the merits of a fully formal specification, defined in a language which is mathematically precise and in which strict syntax and semantics are used, and the need in the early stages of design for a less formal and more intuitive approach as an aid to understanding the system being designed. Carey & Graham (1987) consider that any (design) method must be formal enough to provide benefits throughout the development cycle, and natural enough for designers to use it productively. Green (1986) argues that on the one hand design notations can be very informal, since their main purpose is to record the thoughts of the designer, but that on the other hand the notations used in the implementation of user interfaces must be formal, since they will be used to produce the implementation of the interface. Cockton (1987) sees the need for users to be able to comprehend and understand future dialogue models as more important than identifying a mathematically optimum model. Moran (1981) supports this view, arguing that a model can be too abstract, and thus difficult, for most people to grasp. Harbert, Lively & Sheppard (1990) consider the range of things that must be specified in a user interface to be too broad for any one form of specification.

Guindon (1990) considers that the early stages of design are best characterised as 'opportunistic'. He defines opportunistic design (p.336) as "design in which interim decisions can lead to subsequent decisions at various levels of abstraction in the solution decomposition". The results of this study have implications for methods and environments that support the early stages of design: representation languages should

support a smooth progression from requirements expressed informally to design decisions expressed formally or semi-formally, in code.

Docker (1989) in the context of the development of a structured analysis programming environment, quotes from Gehani & McGettrick (1986, p.vii): "Formal specifications do not render informal specifications obsolete or irrelevant; although they (formal specifications) can be checked to some degree for completeness, redundancy, and ambiguity, and can be used in program verification, they are often hard to read and understand. Consequently, informal specifications are still necessary as an aid to the understanding of the system being designed; informal and formal specifications complement each other". Docker also suggests the possibility of a spectrum of descriptions going from informal at one end to strictly formal at the other, and introduces the term 'semi-formal' to describe, for example, the class of techniques called 'structured systems analysis'. He sees these techniques as having been developed (p.8) "in an attempt to improve both the approach to analysis, and to place the emphasis more on the *graphical* presentation of information as a better method of communications".

Focus on a graphical representation

It has been argued (Tse & Pong, 1991) that graphical languages are better at the higher levels of abstraction, and textual languages are better for providing detailed description. In the context of specifying systems requirements, Tse & Pong argue that graphical representation of complex material is much more comprehensible than its textual counterpart, because it is two-dimensional rather than one-dimensional, and therefore provides an additional degree of freedom in presentation; because it can show naturally both hierarchy and concurrency; because it can be read selectively rather than linearly; because it reduces the number of concepts to be held in short-term memory; and because the reader can move naturally from higher to lower levels of detail. Harel, Lachover, Naamad, Pnueli, Politi, Sherman & Shutl-Trauring (1988) consider that languages for describing reactive systems ought to make it possible to move easily from the initial stages of requirements and specification to prototyping and design. They go on to suggest that specifications, including behavioural aspects, should be based to a large extent on 'visual formalisms' depending on a small number of carefully chosen diagrammatic paradigms. (Harel, 1988, p.528) supports the use of visual formalisms for representing computer-related systems "because they are to be generated, comprehended, and communicated by humans".

This suggests an approach to interface design in which a semi-formal high level graphical specification is augmented by non-graphical detail, and subsequently mapped into a more formal dialogue specification language for implementation; that is, an approach based upon multiple specifications. At the highest level of design what is needed is a visual vocabulary suitable for 'sketching' the interface, a representation able

to capture both user and system interaction with the visual display. In the words of Jacob (1985, p.52): "a visual representation chosen for this purpose needs to describe the external (user-visible) behaviour of the user interface of a system precisely, leaving no doubt as to the behaviour of the system for each possible input. It should separate function from implementation, describing the behaviour of a user interface completely and precisely without unduly constraining the way it will be implemented".

Chakravarty & Kleyn (1990) consider that compactness and readability are crucial to the usefulness of visual formalisms (which they term 'visualisms') applied to the description of the behaviour of reactive software systems. The control aspects of such systems comprise the generation and processing of external and internal events, the actions performed as a result of the events, the directions of control flow, and the concurrency of events, all of which contribute to the difficulty of specifying and understanding system behaviour. In any visual formalism there is a trade-off between the visual complexity of the graphical notation and the effort required to use it. As the visual complexity increases, the effort required to use the notation effectively also increases.

It is interesting to note that the most successful and widely adopted graphical representations in other fields of systems design have been simple notations with a few easily understood constructs, for example, the E-R notation (Chen, 1976) in high level data base design, and the data flow notation of structured systems analysis (Gane & Sarson, 1979). The event model, a candidate for describing direct manipulation interfaces, has to date lacked a graphical representation. Issues associated with graphical notations are explored further in Chapter 3 of this thesis.

1.2 Objectives of the research

The principal objective of this research is to develop a semi-formal graphical notation and associated methodology for use during the high level design phase of interface development. The notation should be targeted specifically at direct manipulation interfaces and should be extensively evaluated in the context of an existing interface. Implicit in this objective are the following secondary objectives:

- That visually, the notation should:
  - have a small number of simple concepts;
  - be suitably expressive, and in particular reflect the object oriented nature of direct manipulation interfaces;
  - be compact, and maintain locality between dialogue elements.
- That the notation should be suited both to the analysis of existing interfaces, and to the high level synthesis of component dialogues for new interfaces.

- That the notation should be executable, and thereby support a limited form of early prototyping of interfaces.
- That the notation should be mappable into a formal dialogue specification language, and thereby support interface prototyping and implementation.

1.3 Approach adopted

The approach to achieving the objectives set out in Section 1.2 can be summarised as follows:

- The nature of direct manipulation interaction is established, and the desktop metaphor is examined with reference to the Apple Macintosh. Three views pertinent to the development of direct manipulation interfaces are identified (Chapter 2).
- Existing dialogue notations are reviewed in the context of direct manipulation interfaces, with a focus on graphical representations for specifying underlying interface behaviour to support high level design. Some preliminary results of this research have been published (Phillips, 1991). Based on this analysis, the required scope of a direct manipulation dialogue model is established (Chapter 3).
- A task analysis of part of the Macintosh interface is undertaken in order to establish the fundamental nature of direct manipulation interaction. The results of this analysis have been published (Phillips & Apperley, 1991). A framework for developing and extending Lean Cuisine (Apperley & Spence, 1989), a tree-structured graphical notation for describing the behaviour of menu systems, is defined (Chapter 4).
- Lean Cuisine is developed incrementally into an object-based multi-layered notation, Lean Cuisine+, providing for specification of the full range of constraints and dependencies which exist between selectable dialogue primitives in a direct manipulation interface. This development is achieved through the description of the behaviour of part of the Macintosh interface (Chapters 5 and 6). Aspects of this research have been published (Phillips, 1992).
- The role of Lean Cuisine+ in interface design is considered, and an orthogonal task layer is added to the notation. A five stage design methodology, commencing with a task decomposition is developed and applied to an example interface (Chapter 7).
- A specification for a software environment to support the construction, browsing, and execution of Lean Cuisine+ specifications is developed. The extensions to support execution complete the Lean Cuisine+ notation. The software environment is partially prototyped using HyperCard. Mapping to a more formal dialogue specification language to support interface prototyping and implementation is considered (Chapter 8).

- The Lean Cuisine+ notation is reviewed and the contribution of the research examined critically (Chapter 9).

1.4 Structure of the thesis

The thesis is arranged in three parts. Part 1 includes this introductory chapter, Chapters 2 and 3, which present the major part of the literature survey, and Chapter 4, which establishes a framework for the notation developed in Part 2.

In Part 2, the basic Lean Cuisine+ notation is developed in Chapters 5 and 6, and extended in Chapters 7 and 8. Chapter 7 also presents a methodology for the construction of Lean Cuisine+ specifications, and Chapter 8 defines a software environment to support the notation.

Part 3 contains a single review chapter, Chapter 9, in which the contribution of the research is examined and further work identified.