A Functional Model for Graph Interaction

Simon Brown

M. Phil.
University of Edinburgh
1991
Abstract

This thesis presents a new method for the construction of general interfaces, the Stream, and describes its use in the building of an interface for manipulating graphical structures. It also describes XGE, an implementation of a structural graph system based on this design.

A functional hypergraph is built up hierarchically from three types of basic objects, where each object is described in terms of collections of functions for evaluating attributes. Such a graph is composed of a closed network of specifications. All aspects of a graph object are expressed in terms of general attributes, and can thus be manipulated in a uniform way.

A streams interface for such a graph operates by transforming the attributes of objects. A stream acts orthogonally on a graph to produce a transformed image graph, which is also structured as a closed hypergraph of functional specifications.

An interface can then be constructed by permitting modifications to the attribute specifications in either the original graph, the topmost graph in a stack of stream graph images, or to the stream specifications which make up this stack. Any such modification generates a cascade of functional specification changes, which produce the appropriate transformations to maintain consistency.
Acknowledgements

I am grateful for the help and encouragement from my supervisors: Joachim Parrow who supervised me during the first year of this work, and Eric McKenzie who became my supervisor when Joachim left the university.

I would also like to thank Professor Robin Milner, who was my secondary supervisor for the duration of the work for this thesis.

Thanks go to Dr Maggie Cooper and to Paul Dourish, who proof-read several parts of the thesis. Any remaining errors are entirely my responsibility.

The first two years of the work for this thesis were funded by International Computers Ltd. I would also like to thank the Laboratory for Foundations of Computer Science, and the Department of Computer Science at Edinburgh University, who helped to fund the work for the subsequent years.
Declaration

All the work in this thesis is mine, except where stated otherwise. I composed this thesis myself.

Simon Brown

September 1989
# Table of Contents

1. **Introduction** ........................................ 10  
   1.1 Overview ................................................. 10  
   1.2 The Streams Model for Graphs ............................ 11  
   1.3 Motivation .............................................. 13  
   1.4 Implementation ......................................... 13  
   1.5 Structure of Thesis ..................................... 13  

2. **Overview** .............................................. 15  
   2.1 A Streams-based Graphical Interface ..................... 15  
      2.1.1 Streams .............................................. 15  
      2.1.2 Graphical Structure .................................. 16  
      2.1.3 Structural Streams ................................... 17  
   2.2 Conventional Graphical Interfaces ....................... 18  
      2.2.1 WYSIWYG .............................................. 18  
      2.2.2 Views and Folds ...................................... 18  
   2.3 Concrete Examples ....................................... 20
Table of Contents

2.3.1 Zooming and Panning ........................................ 20
2.3.2 Detail-Hiding ................................................ 20
2.3.3 Virtual Objects ............................................. 21
2.3.4 Addressing .................................................. 21

3. Background ..................................................... 22
  3.1 The Andrew System ......................................... 22
  3.2 UNIX Streams ............................................... 23
  3.3 Functionalization .......................................... 25

4. Graph Implementation .......................................... 30
  4.1 Introduction ............................................... 30
  4.2 Attributes .................................................. 32
  4.3 Representation of Attributes .............................. 33
  4.4 Interaction ................................................ 34

5. Transformational Interface Concepts ......................... 36
  5.1 Structure of an Interface .................................. 36
  5.2 Domain-Based Interfaces .................................. 37
  5.3 Transformations ............................................ 38
  5.4 Object Referencing ....................................... 40

6. Graph Domains ................................................ 42
  6.1 Structure .................................................. 42
6.2 Attribute Control ........................................ 42
6.3 Concrete Attributes ..................................... 45

7. Implementation of Graph Domains .................. 48

7.1 Datatype Representation ................................ 48
7.2 Design .................................................. 49
  7.2.1 Descriptors ......................................... 49
  7.2.2 Types ............................................... 50
  7.2.3 Geometry Values .................................. 50
7.3 Details .................................................. 51
7.4 Attributes .............................................. 55
7.5 Labels ................................................... 60
7.6 Implementation Interface .............................. 62

8. Implementation of Graph Filters .................. 67

8.1 Introduction ........................................... 67
8.2 Message Formats ....................................... 68
8.3 Filter Types ........................................... 70
8.4 Addressing ............................................. 72
8.5 Filter Action ........................................... 73
8.6 Filtering Summary ..................................... 74
8.7 Filter Action (continued) ............................. 75
8.8 Filter Stack Operations .............................. 76
# Table of Contents

9. Implementation of Graph Displays 78
   9.1 Purpose ........................................... 78
   9.2 Graph Structure ..................................... 79
   9.3 Window Structure .................................. 81
   9.4 Views ............................................. 82
   9.5 Graph and Window Operations ......................... 85
      9.5.1 Graph Operations ............................. 85
      9.5.2 Window Operations ............................. 87
      9.5.3 Viewer Operation .............................. 88

10. Implementation of an Interactive System 89
    10.1 Control Messages .................................. 89
    10.2 Production of User-Level Control Messages .......... 90
       10.2.1 Input Events ................................ 90
       10.2.2 Menu Bindings ................................ 91
       10.2.3 Key-Event Bindings ......................... 92
    10.3 Low-Level Interaction Operations .................... 93
    10.4 High-Level Interaction Operations ................... 95
    10.5 Object-Based Interaction ........................... 95

11. Appendix A: Examples and Applications using XGE 100
    11.1 Summary .......................................... 100
    11.2 Example Implementations of Common Idioms ............ 101
Table of Contents

11.2.1 Menus ............................................ 101
11.2.2 Folds .............................................. 102
11.2.3 Highlighting ...................................... 103
11.3 General Remarks ................................. 105
  11.3.1 Object Oriented Methods ....................... 105
  11.3.2 Functional Methods ............................. 107
  11.3.3 Generating Displays .......................... 109
  11.3.4 Selection ........................................ 113
11.4 Applications ........................................ 115
  11.4.1 Process Modelling ............................. 115
  11.4.2 Harel Formalism ............................... 116
  11.4.3 Imposition of "Views" ......................... 117

12. Appendix B: Extensions to XGE ........................ 121

12.1 Limitations of the Current Implementation .......... 121
  12.1.1 Design Limitations ........................... 121
  12.1.2 Implementational Limitations .................. 122
12.2 Redesign Issues .................................... 124
12.3 Optimization ...................................... 126

13. Conclusions ........................................... 128

13.1 Summary .......................................... 128
  13.1.1 Graph Representations ....................... 128
### Table of Contents

13.1.2 Filter Streams .................................. 129
13.1.3 Graphical Display .............................. 129
13.1.4 User Interface Specifications .................... 130
13.1.5 Construction of Interactive Graphical Systems .... 130

13.2 Remarks ............................................ 131

13.2.1 Advantages to the Approach ..................... 131
13.2.2 Disadvantages to the Approach .................. 132
13.2.3 Implementation and Language Issues ............. 132
Chapter 1

Introduction

1.1 Overview

This thesis presents a new form for the construction of general user interfaces, the stream, and describes its use in the building of an interface for manipulating graphical structures.

A functional model for graphical structure is given, based on a state-machine form of architecture similar to the process-model of CCS[8]; then a streams interface is built over this model.

Graph Structure

A graph is represented as a collection of objects. Three types of objects are involved, each corresponding to a particular aspect of the structure of the graph; "node objects" act as the nodes or points in the graph, "arc objects" act as arcs or edges between nodes, and "port objects" act as junctions between nodes and arcs.
This gives a flat model of a graph; it has no additional structure. Structure is provided by allowing objects to contain sub-objects. Thus, each node in a graph can contain a collection of nodes — a subgraph. For completeness, the other types of objects — arcs and ports — can also contain subobjects. This gives a hierarchical model of a graph, as a collection of embedded objects arranged in a tree structure from a single "root" node object.

Visual aspects of a graph are represented at the object level; each object contains information describing how it is to appear on a graphical display.

A graph is a dynamic entity — objects can be created or destroyed, or moved around.

1.2 The Streams Model for Graphs

The main part of this thesis deals with a method by which such graphs can be manipulated, to provide an interactive graph editor. A flexible method by which a graph can be displayed graphically is presented, based on the idea of "streaming", to provide a user interface.

A user interface is built up by allowing "modified versions" of a graph to be viewed in a physical display. The details of the modifications are under the control of the user, so that the display corresponds to a user-controlled image of the graph.

This image is built up from the "real" graph by applying successive regular transformations to its objects, each transformation producing a slightly more distorted copy of the graph. The final transformation produces the copy of the graph which is displayed visually. The user interface is then in place — if the user makes any changes to the visual graph, then these changes cause transformations to occur, streaming down through the stack of distorted graphs, until eventually they cause
an appropriate alteration to occur to the "real" graph. Similarly, if the real graph should change, a stream of transformations is sent up through the stack of distorted graphs, until it reaches the uppermost image, where the appropriate alterations are made and displayed visually.

The system model can be considered as a stack of graphs, connected by transformation streams. The bottom-most graph is the "real" graph, under the control of some program. The top-most graph is the "visual" graph, under the control of a user. A change in either will generate a stream of transformations which result in an appropriate change in the other.

A streams interface for a graph operates by transforming the attributes of objects. Since a graph is described entirely by attributes associated with its constituent objects, this is enough to give a description of a new, "deformed" copy of a graph.

Each individual transformation acts by altering the specification of an individual object, making a new copy in its "image" graph. A form of pattern-matching is used to decide which objects are to be effected by any particular streams transformation module.

As new modules are inserted between the "real" and "visual" graphs, the feel of the interface is altered. Modules may be inserted or deleted at any time, under the control of the user, providing a programmable user interface.

A graph editor interface can thus be constructed by allowing the user to cause changes in the visual graph, and in the stack of stream transformation modules, and allowing a "driver" program to cause changes in the underlying graph, as shown in the diagram overleaf.
virtual graphs

user interface

filter streams

physical graphs

driver interface
1.3 Motivation

The work in this thesis is an attempt to describe an interactive user interface system in a structured way. A new user interface methodology is presented, based on structural transformation of data.

1.4 Implementation

A particular implementation of these ideas is described — the XGE system. This is a partial implementation, in that many of the generalities that follow from the above description are provided in only a restricted form.

XGE is a general-purpose graph-editor. It is a "template" system, in the sense that it provides a sufficiently general programmable base on which specific editors can be built. It can be loosely thought of as a specification language for building user-interfaces for manipulating structural objects.

The implementation is written in the Standard ML programming language, and provides graphical support through the X windowing system.

1.5 Structure of Thesis

Chapter 2 gives a general overview of the contents of the thesis, in rather more detail than in this introduction.

The following five chapters describe an abstract model for an interactive graphical system, based on streaming. The structure of graphs is described in general
terms in chapter 4. Some background on transformational systems is given in chapter 3. Chapter 5 describes how transformational streams may be imposed on graph domain structures. A description of how this can then be used to produce an interactive system is given in chapter 6.

The next four chapters describe, in some detail, a particular implementation of this model, the XGE system. Details of the implementation of graph structures are given in chapter 7. Chapter 8 describes how filters (ie, stream channels) are implemented. A description of the physical mapping to a graphical display is given in chapter 9. The means by which interactive systems may be built out of XGE are described in chapter 10.

Chapters 11 and 12 give examples of how the system may be used to various ends, and how the current XGE implementation could be extended and improved.

Conclusions drawn from this work are listed in chapter 13.
Chapter 2

Overview

This chapter gives an overview, or extended introduction, to the various concepts to be explained in more detail later in the thesis, and contrasts them with more conventional methods.

2.1 A Streams-based Graphical Interface

This section describes how a streams-based interface works, and the advantages of such an approach. It then describes a functional method for modelling graphical structures. Finally, it describes how a streams interface may be imposed over such a model to provide a reasonable graphical user-interface.

2.1.1 Streams

The streams paradigm for viewing of structural objects is based around an encoding of the structures involved, together with an encoding for transformations to be applied to the structures. A stream is a channel for communication between one structure and another. Such a channel is composed of a chain of separate stream
modules. Each of these modules performs some form of transformation on incoming structure descriptions, producing distorted images of the original descriptions, which are then passed on to the next level.

Streams are a viable vehicle for graphical interfaces, as they allow the natural imposition of *views* by means of structural transformation. A Graphical system may be built up by associating the bottom-most structure with an internal model, and associating the upper-most with a graphic display.

Streams are particularly suited to *dynamic* systems, where any views may be modified on demand from the user interface. This can be done by considering a chain of streams modules as a list — modules may be inserted or deleted at any point. This is a generalization of existing stack-based streams implementations\(^1\) where the streams are dynamically alterable by use of push and pop operations.

This definition of the transformation model does not specify in any great detail how the intermediate levels are represented. One obvious choice is a wholly lazy representation, where a structure is only produced when required, in response to incoming stream data, and is destroyed as soon as the structure has been passed on to the next level. Alternatively, an eager representation would keep copies of all the intermediate structures; they can be thought of as a cache.

### 2.1.2 Graphical Structure

A graph is represented as a network of object structures. Each such object is identified by a unique descriptor, and contains a collection of attribute definitions. Each attribute is represented in the form of a functional\(^2\) expression, together with

---

\(^1\)ie, the various UNIX implementations of stream-driven I/O.

\(^2\)The XGE implementation is written entirely in the *Standard ML* language
a list of "dependent" objects, on whose value the expression depends. All aspects of an object are dealt with in terms of attribute values; examples of attributes include: position, size, appearance, structure, etc.

A graphical system (consisting of an active graph — a graph which can react to events, rather than remaining static) is thus expressible in terms of the "programming" of each component object. Since this programming — i.e., the attribute value expressions — can itself involve the creation of new objects, each with its own programming embedded within it, it is possible to produce completely dynamic active graphical systems; the entire structure of the graph can be automatically self-modified in reaction to events.

In structural terms, a graph is built up from node objects, arc objects (links between nodes) and port objects (communication-points acting as an interface between nodes and arcs).

### 2.1.3 Structural Streams

A streams interface can be imposed over such a general graph structure by producing streams modules which can manipulate the source attribute specifications to produce new target attributes. Thus, for example, a module whose purpose is to hide the internal structure (i.e., subobjects) of a particular object, need modify only the specification of the subobjects attribute — everything else passes through the module unaltered.

Thus, a stream module would act on incoming specifications from the source graph, to produce new specifications which are passed on to the target graph. The bottom-most graph hold a 'true' data structure, whereas the upper-most graph hold a particular graphical representation of this structure.
Chapter 2. Overview

To this end, the system provides a number of different types of modules — modules which can create new objects from old ones, modules which can alter existing objects, and modules which can delete objects. Note that any alterations made by a module will apply only at the "target" of the stream in question — the original graph remains unaltered; a traditional, non-streams-based interface is provided for updating objects within a graph.

2.2 Conventional Graphical Interfaces

2.2.1 WYSIWYG

The most pervasive form of graphical interface is the WYSIWYG paradigm. This is not particularly useful when the main goal is the representation of structure — the entire point of WYSIWYG is its lack of an underlying structural basis.

2.2.2 Views and Folds

The conventional means for providing alternative viewpoints for structures is by the use of views and folds. These allow the underlying structures to be viewed in distorted ways. However, the lack of structure within the specification of a view itself forces such a view to contain detailed knowledge of the basic structure of the underlying graph. Thus it is not immediately extensible to arbitrary graphs; a new view must be completely redesigned for every new aspect to be viewed from an arbitrary structure.

\[^3\]"What You See Is What You Get" — the form of interface in which the external and internal representations of structure are identical.
Traditional views suffer from the following drawbacks:

- **Non-reusability.**
  A view must, by virtue of its very nature, possess detailed knowledge of the structure of the graphs being manipulated. Streams, in contrast, provide a general mechanism, somewhat akin to the concept of *pattern-matching*, by which arbitrary structures may be transformed in consistent ways. Thus, a view-based system must be completely re-designed before it can be applied to a new form of graph structure — whereas a streams module can be simply (in principle at least) re-used as it is.

- **Non-extensibility.**
  Once a view has been designed, it is somewhat fixed — it is a non-trivial matter to extend it to include additional viewing schemes. This, also, is due to the hard-wiring into the view mechanism of the details of the structures being viewed. In theory, this should not necessarily be the case; but any standard implementation of views will suffer from this deficiency.

  In contrast, a streams module can be easily extended to provide additional transformations, in a way analogous to the addition of extra clauses in a pattern-matching expression.

  Of course, these problems are by no means insurmountable. Polymorphism and inheritance of types can be used to make views more generally applicable. These correspond to a "programming in types" approach, as opposed to the "programming in functions" approach of the streams method.
2.3 Concrete Examples

The following sections describe briefly how many features and mechanisms which could be required of a graphical interface may be implemented in terms of a structural streams architecture. It by no means pretends to be a complete list of capabilities.

2.3.1 Zooming and Panning

A graph may be zoomed-in on or panned-out over by means of a trivial stream transformation. A graph is viewed by means of the aiming of a graphical window onto the graph by means of a stream. A window is a means of providing a physical graphical display. It contains several graph-independent attributes, packaged in a Viewer structure. These attributes include zooming/panning information. Thus, these can be obtained simply by tuning the appropriate parameters in the window; no direct stream action need be taken at all.

2.3.2 Detail-Hiding

Internal details of a graph (such as subobjects, labels, diagrams, etc) may be "hidden" in a display by means of a detail-hiding streams module. Such a module acts as a filter to prevent the passing of these details down the stream. Thus, a module which converts the subobjects list attribute of a given object to an empty list will result in the non-display of any subobjects of that object when viewed in a window through a stream which contains this module.
2.3.3 Virtual Objects

"Virtual" (or "combined") objects are dummy objects which stand in place of some arbitrary collection of objects — this is really just another kind of detail hiding. A streams module may produce virtual objects by replacing this object collection with a new object. The new object will have attributes defined by the module itself — so these may well depend on attributes from the constituent object collection.

2.3.4 Addressing

Each of the examples given above could easily be implemented in terms of a more traditional 'views' mechanism. However, the pattern-matching form of addressing that stream modules provide would allow these to be easily extended into more interesting operations.
Chapter 3

Background

I will compare the transformational model with two other ideas — the object-oriented principles as used in CMU’s Andrew system, and the almost-functional principles underlying the UNIX Streams mechanism.

3.1 The Andrew System

Andrew [9] is a completely object-based system, implemented in a superset of the C language. Its fundamental idea is that of having a set of basic objects which are manipulated through a sequence of views. Things such as windows, scrollbars, boxes, etc, are all instances of views, whereas their contents are instances of objects — the views are a means of combining the objects for the purpose of interaction. Each view is in communication with its parent view and all its children views (thus building up a tree of views over the set of objects). Each view handles an event by either calling the appropriate method function for dealing with that event, embedded in its definition, or else farming it out (by effectively forwarding the event message) to its parent or child views — or possibly a combination of these, in cases where it must do some work itself which will in turn affect other views.
This method allows input and output events to be handled almost identically, as all that need be done at each level is to decide whether or not to handle an event. Input events come in at the root view, and gradually trickle down until they reach a view which is prepared to handle them — if no view handles them, then they are forwarded to the actual objects themselves. Output events are produced by the objects and introduced into the view tree at the leaves, and move towards the root until they find a view willing to handle them. If none does, then they will reach the root (i.e., the window managing system itself).

The advantages of such a system are obvious; each view need know only how to handle those events which actually concern it, as unknown events can simply be passed on unchanged to other views. However, it has the disadvantage that the message-passing is fixed and static — each view must know by name each of the other views, so making any modifications to the structure of the view tree causes problems. More importantly, it is inherently inflexible in that no levels of meta-events can occur — the object-oriented approach causes the message-passing to be transparent, and thus not open to manipulation.

This transparency arises from the very basis of the object-oriented paradigm. An object performs actions by responding to a message, where a message is solely a request for the action to be performed. If viewed as the calling of a "local" function particular to the object, the limitations of this schema can be seen.

### 3.2 UNIX Streams

Streams [12] are a mechanism for generalized I/O control, developed for the internal Bell Labs UNIX systems, Editions 8 and 9. The basic concept is that of stacking modules on top of an I/O connection, where each module is responsible for some
kind of processing. Since the I/O "packets" are directly analogous to message-passing, this seems a particularly useful scheme for much more general purposes.

Each module consists of a pair of filter queues, one in each direction (i.e., one for incoming messages and one for outgoing). The processing associated with each filter has access to both queues, so messages may be produced internally by the filters and sent either forwards or backwards, independent of the direction of the original message. The diagram overleaf shows how multiple stream modules may be combined to produce a single transformational stream.

If the messages being passed are interpreted at the ends as being events targeted on objects, then this schema can be directly transferred over to the arena of interface systems.

Any form of I/O control is encoded in terms of special control packets which can then be sent down a stream in the same way as data is sent. Each stream module examines the control packet to see if it can be handled at that level. If it can, then the appropriate changes to the module occur and the packet goes no further. Otherwise it is passed through to the next module in the stream stack. Thus, a control packet that does not address any pushed module will pass through unchanged to be handled by whatever is reading from the other end (software or hardware).

The means by which messages can pass through a sequence of stream modules is shown in the diagram overleaf.
There are certain annoying restrictions in all existing implementations of streams (both in the Bell Labs "Ritchie Streams" and the AT&T "System V Streams"). The most damaging of these is that only a small set of "known streams" may be pushed — new stream modules may not be produced dynamically.

Another restriction arises from the UNIX concept of data — the "everything is a list of bytes" streaming concept. This is all very well for hardware device I/O, but for more structured uses a typed-stream mechanism would be invaluable. This cannot be overcome simply by providing structure-recognizing modules, since the data used at the target end of the stream must not have its structure broken down.

Another problem arises from the purely stack-based principle of streaming. This prevents a stream module from being inserted into the middle of an existing stack of modules. Again, given the purposes for which they are intended in the UNIX domain, this is not too important (and improves efficiency).

### 3.3 Functionalization

The framework of the Andrew system can be easily made functional simply by re-placing its transparent object-oriented message-passing with a streams-based framework.

It can also be made less "flat" by imposing a new dimension of view-placing, whereby a meta-view may be placed over an existing view to give a "view of a view", thus allowing views to be manipulated in exactly the same way as objects — an object becomes merely an end-point in the tree. The tree model is not really much good any more, as multiple views can be imposed on a single object, and multiple objects can be scanned by a single view.
Chapter 3. Background

In principle, there is no real limit to how many levels of meta-viewing may be implemented — the only problem lies in designing the view structure so that all levels of message-passing are consistent, so that any view can be put onto any object (be that a view object or a basic object). In practice, views will have all kinds of user-interaction code built into them, so this isn’t as powerful as it might at first seem — it would be better if the unpleasant “user dependent” features could be moved out into exterior modules, leaving the views uncluttered with such things.

Streams modules may be constructed from a set of basic atomic stream-actions, by defining the transformations which may occur. The transformations must, of course, be defined in both directions, so that the stream may act as a bidirectional filter. The complexity involved in the structure of such a module is limited only by the expressive power of these basic actions. Needless to say, the current implementation does not provide much expressiveness, which somewhat limits the types of modules which may be produced. In practice this is not a handicap, as the restrictions imposed by this only limit the modules to being “uniform”. (Actually, this is not quite the case — it is possible to produce slightly non-uniform filters, so care must be taken when designing a filter to ensure that it is in fact uniform.)

This concept of uniformity, or continuity, as applied to a streams module, can be expressed in terms of the commutativity of the diagrams overleaf.
virtual graph → induced virtual graph

base graph → new base graph

filter stream → induced modification → filter stream
virtual graph \rightarrow \text{virtual modification} \rightarrow \text{new virtual graph}

filter stream

base graph \rightarrow \text{induced modification} \rightarrow \text{induced base graph}
An alteration in a base graph must give rise, via filtration of this alteration, to an equivalent alteration in its filtered image; conversely, an alteration in the image must give rise to an equivalent alteration in the base, through the filter. This is made more complicated by the third commutation: an alteration of the stream filter itself must give rise to an alteration in the image corresponding to virtual alterations in the base. The diagrams sum it up quite nicely.

The details of an "ideal" implementation can be inferred directly from this diagram. A summary of the ideas involved follows.

- **Base-Graph Alterations.**
  If a base graph is altered in some way, then the alterations are passed through the filter in order to induce equivalent alterations in the image. This is relatively straightforward — if all filters involved are uniform, then only those objects which were changed in the base can give rise to changes in the image, thus only the altered objects need be re-filtered.

- **Image-Graph Alterations.**
  If an image graph is altered, then these alterations must be reverse-filtered to induce equivalent alterations in the base from which the image is derived. This is merely the inverse operation to the Base-Graph case, and is similar in most respects.

- **Filter Alterations.**
  This is the most difficult case to deal with efficiently. Filters can be altered in any one of three ways: addition ("pushing"), subtraction ("popping") or internal modification. I will deal with each case separately.

  - **Addition.**
    When a new filter module (or set of modules) is added to an existing
filter stream, a set of alterations must be generated to change the image-
graph to bring it into line with the new filter stream.

The simplest, but least efficient, method for achieving this is to simply re-
filter the entire base graph through the new filter. More optimal methods
can be used if the filters can be analysed to detect which objects are to be modified by each filter module. In this case, objects which could have been so altered can be reverse-filtered back to the position at which the new module is to be inserted, then forward-filtered through the new module and onward back to the image graph.

This is a lot more difficult than it actually sounds, as the objects to be put through the new filter do not necessarily exist until the reverse-
filtration has occurred — in a lazy implementation, they are purely tem-
porary, mid-stream objects — thus it is next to impossible to predict which objects in the image need to be reverse-filtered.

- *Subtraction.*

Removal of filter modules from an active stream is similar to addition, but does not lend itself even to that level of optimization that addition does, in terms of reverse-filtering only of necessary objects.

This is because of *combination* filters. Straightforward "linear" (ie, structure-maintaining) filters can be simply reverse-filtered in an ob-
vious way, but the overheads involved when multiple objects must be unfolded during this process are quite daunting.

- *Internal Modification.*

This form lends itself most easily to optimisation, as the details of what has changed are more readily available for analysis. For example, if the only change to the filter module is in the ‘names’ of the objects to be acted on, then the image graph can be re-evaluated by reverse-filtering only those objects which have any of these target names as ancestors,
back down to the modified filter, and then forward-filtering to produce new attributes in the top-most graph.

the XGE implementation does not take advantage of any of the possible optimization schemes.
Chapter 4

Graph Implementation

4.1 Introduction

The types of graphs I am interested in modelling are essentially hierarchical hypergraphs, though for most purposes I'll stick to ordinary "binary-edged" graphs. The overall idea is to model graph-structure in as simple, but also as complete, a way as possible, given the requirements which must be met. These requirements can be summed up by the need for consistency of design, so that the "functional-addressing" technique can be used to the greatest effect, as described in the overview.

A flat graph consists of a collection of objects, with a collection of relations defined on them. These can be modelled using two types of object — nodes and edges. Each edge is associated with the pair of nodes which it relates, and each node is associated with a set of edges representing relations from or to it. In fact, I can quite easily extend this representation to cover flat hypergraphs, by allowing an edge to be associated with an arbitrary number of nodes, rather than restricting it to exactly two.
Chapter 4. Graph Implementation

The meeting-place between a node and an edge is represented by a third kind of object — a port. This is a useful abstraction in that it allows the formation of arbitrarily complex “junctions”, possessing whatever kind of internal structure is desired, without sullying either the node or the edge. It has the fortunate effect that the existence of such objects hides all information regarding the structure of these junctions from the both the node and the edge, giving the whole model additional symmetry — if they were not present in the representation, then junction information would have to be stored explicitly in either the node or the edge (giving asymmetry), or in both (giving redundancy).

It is useful to separate the functionalities of the three types of object (rather than only having nodes, and maybe edges), as it allows transformations on objects to act entirely on a single aspect of the graph.

The hierarchical requirement can be met by associating with any object (node, edge or port) a collection of subobjects of the same type. Thus, a node may “contain” a collection of subnodes, an edge a collection of subedges and a port a collection of subports (and so on, recursively). A hierarchical graph is then identified by a single node — the root node of the graph — since all other nodes can be reached from this.

The subobjects of a node correspond to a subgraph. The subobjects of edge and port objects are less obvious. If an edge contains subedges, then that main edge can be considered as being a ‘cable’ of edges, which can be manipulated together as though they were one. Similarly, subports allow many port objects to be manipulated together.

This model raises two problems. First, although it models the hierarchical nature of such graphs extremely well, it presents some difficulties in attempting to deal with properties which are not built up hierarchically, such as the association of edges with nodes (via ports). It should be possible to associate any pair (or
collection, for a hypergraph) of ports with any given edge, rather than restricting the relation to ports within nodes at a particular level of the hierarchy. The obvious solution is not to introduce edges until the hierarchical structure has been produced, and then define them axiomatically as relations between any pair (or collection) of ports.

The second problem is that although this describes the structure of such a graph, it says nothing of how such a graph should be interpreted — there is no means of associating arbitrarily structured data with a graph. The solution to this would seem to be to associate with every object a collection of data components. These components can contain arbitrary data, though it seems reasonable for this to include the possibility of containing objects of any of the three types.

The main difficulty is in resolving what type of entity these data components should be. Ideally, they should have the potentiality of containing any kind of data at all. The name "data" is perhaps misleading; their true purpose is to provide "methods" (in the object-oriented sense) for responding to events.

4.2 Attributes

Each of these "associativities" on a graph object is just an attribute of that object, which build up a specification of the overall structure. Ideally, these should specify all of the hard-wired attributes — any others will be inherently interpretation-specific, and so will be part of the data components (and thus not available for inspection at the graph-structural level).

An alternative, more radical, view is that this structural information is as much part of the data as any other attributes, so they should all be data attributes — thus effectively generalising the "internal" attributes into this common interpretation-
dependent data pool, so that there are no explicit hard-wired interpretations, even for the underlying graph structure. This would have the advantage that the apparent structure of a graph would be accessible only through the appropriate method-calls of its constituent objects (and so would not be "fixed" in any sense — the apparent structure could depend on the environment from which it is observed).

In the XGE implementation, a combination of these is used. A few ‘key’ global attributes are associated directly with an object, rather than being dealt with as part of its data components. This corresponds to imposing built-in interpretations on these attributes, while leaving other, more data-dependent, attributes at a more general, only indirectly accessible, level.

4.3 Representation of Attributes

The best form for attributes to take would seem to be a dependency-based one. Each attribute is represented by the following specification information:

• an evaluation function, which produces the current value of the attribute when instantiated at the current environment.

• a dependency list, consisting of references to attributes in other (or perhaps even the same) objects on which this value depends. (It is this list, more than anything else, which determines the "environment" within which the evaluation function is called.) When any such attribute named in this list is modified in some way, then the evaluation function for this attribute must be called.

This may easily be generalized by including a regeneration function component in an attribute specification, this being a function which returns a new evaluation
function and dependency list, once some event has occurred which necessitates this change (such as an “alter-attribute” event, to pick a trivial example). Any number of meta-levels of specification may be implemented by the provision of suitable meta-regeneration functions, similarly. Alternatively, a single regeneration function could be provided, with a list of meta-dependencies — each element in this list corresponding to another meta-level.

Provision of regeneration functions gives a simplistic language in which a graph structure may be programmed to respond to events, in a way analogous to a state machine. A graph that has been programmed in this way can then act as its own, stand-alone driving application. Applications that cannot be easily modelled as a state-machine can drive their graphs explicitly from without.

4.4 Interaction

Interaction with a graph involves only the sending and receiving of control messages to and from named objects of that graph. Incoming event messages are produced elsewhere (in another graph, or from a user-interface), and are directed to whatever objects they are aimed at (having been first translated into the local graph addressing context). This results in the method-functions of those objects corresponding to these events being called, and responding in whatever way is appropriate (possibly by generating a new batch of event messages, aimed either within the same graph, or out to some other domain).

Note that this treatment of events is “flat”, as opposed to the hierarchical event-handling of the domain system itself. An alternative arrangement could be to treat the internal graph system as just another subtree in the overall domain system - so that event messages would travel up and down the graph, communicating with other domains via the root, or via domain references embedded within objects’
data-components. Such a system would be much more general — a "graph" would be just another domain type, treated just as any other — but seems inappropriate for a system whose main function is to act as an interface to graphs; this being the case, there seems nothing wrong with the structure of a graph being handled specially and differently from the levels of "meta-systems" imposed above.
Chapter 5

Transformational Interface Concepts

5.1 Structure of an Interface

I will deal with interfaces to systems based on the hypertext model, in which heterogeneous collections of data objects exist in some internal namespace, and have a hierarchy of views placed above them.

The conventional way to describe such systems is to use an object-oriented interface language, in which these collections of data objects are simply particular instances of some fundamental underlying object class, and thus any such object comes provided with a collection of methods for dealing with various events, as defined by the object-oriented class inheritance system.

An interesting thing to note is that there is a considerable amount of functional behaviour behind the scenes in this, but it is never made explicit — the calling of a local method function within an object is comparable to the operation of calling a global method function on that object, after applying the appropriate transformations corresponding to the object-based message-passing.
Chapter 5. Transformational Interface Concepts

The model can be greatly simplified by replacing this object-based class system with a single object type to which all objects belong, and everything can be expressed in terms of transformations between objects. Of course, this could be done without destroying the class system, but there seem to be no very good reasons for doing so, except when the system being modelled is itself class-hierarchy based.

5.2 Domain-Based Interfaces

The types of interfaces with which I shall deal are those which involve the invocation of some kind of user actions to cause changes in the structure of some internal data, in an internal domain. The interface itself lies in the mapping of this internal structure onto a collection of external structures. Interaction is thus made completely symmetric; user-controlled actions take place in the external domain, and system-controlled actions take place in the internal domain. The mappings between these sets of domains take the form of pairs of transformations (which it is convenient to consider as being forms of "full-duplex" two-way transformations, in that each pair represents a transformation and its inverse). Each such pair forms a stream.\(^1\)

This is all reasonably similar to the standard object-based model, except that the message-passing has been made explicit; the mapping together of objects in the internal/external domains is now implemented by defining a transformation

---

\(^1\)In many cases, the inverse transformation can be automatically deduced, but the general functional terms make this impossible in general. Thus, each 'side' of the stream must be specified separately; a stream implementor is required to ensure that they are true inverses of each other, to defend against inconsistency.
between them, rather than by each containing a reference to the other. This is desirable for several reasons:

- the association is symmetric — and necessarily so.
- the association is context-free, in that a transformation is expressed, so far as is possible, in a way that does not tie it down permanently to any particular pair of objects.
- it is possible to define universal transformations, where a single transformation expresses complex relationships between whole groups of objects, rather than having to define each individually.
- the functional nature of the scheme can provide simpler access to parallelism, presuming that some form of independence can be imposed on the various attributes of objects — this could lead to vastly improved performance, were parallel programming constructs to become available for the Standard ML language.

5.3 Transformations

The basic "messages" produced by objects are similar to those produced in the object-oriented system, with the difference that they are undirected — a message is simply a description of actions to take, not of the objects which are to be acted on. A transformation is a function which acts on streams of such messages, addressing them to appropriate target objects on which they should act. There are three obvious places where transformations can be used:

- between the internal data object domain and the internal system itself. This allows the internal domain to exist only in a relative way — it may in turn be
based on some even-more-internal domain from which the system messages originate, and that in turn may be based on another, and so on.

• between the internal domain and the external domain. This is the most interesting part, as it allows definition of whole interface systems to be imposed on top of existing static object domains. Of course, it is really only a special case of the first situation. In fact, it is possible that it may not exist at all, with the internal and external domains being the same, though there are advantages in maintaining a distinction between the "outside" and the "inside" of an interface.

• between the external domain and the "user". Actions by the user can all be converted into appropriate messages which are targeted into the external domain. Inserting a transformation module between the user and this external domain seems a reasonable thing to consider, though it in turn is probably most easily thought of as being yet another special case of the first situation — it corresponds to adding a new outer shell to the interface system.

It is obviously important to design an appropriate format for messages. It is easiest to consider an object as being simply a collection of "attributes", and then a message acts on an object simply by altering a particular attribute. Thus, a message should have packaged together, subject to some naming schema:

• the name of the "target" object (or objects — broadcasting is no problem).

• the name of the attributes to alter (again, broadcasting within the scope of the targets seems a reasonable thing to allow).

• and finally a specification of the actual change.
This last is the most difficult, but can be simplified by enforcing a single "form" for attributes to take. A completely general implementation would treat objects merely as lists of named attributes, with action-functions associated with each (as that attribute's response to a message). A more restricted implementation, where the basic form of the structures is known in advance, is easier could instead provide a fixed set of attributes, with hardwired "meanings".

In fact, there is no real need for an attribute-based system at all; an object can be defined as being simply a function mapping lists of "input messages" (received from other objects or from interaction) to lists of "output messages" (to be sent to certain other objects in response to these inputs). Implicit in this is the assumption that an object has some kind of internal "state" information associated with it, and it is very difficult to escape from such a requirement — truly functional interfaces seem pretty unusable, due to the intrinsic nature of interaction and i/o.

5.4 Object Referencing

One big problem with this is that of identifying objects. To send a message to an object, you need to have a name or handle with which to refer to that particular object. This naming scheme should be consistent over message transformations — if a message causes some state-change in an object, then a transformation of this message should cause a appropriate state change in those objects in other domains linked to it via this transformation.

In addition to referencing objects by name, it is quite often useful to reference objects by context — such as being able to refer to "all objects contained within object α" (in a system where "contained within" is a recognized concept). Here is where a functional approach wins again; such things could be done using functions parameterized by relation-functions.
Chapter 5. Transformational Interface Concepts

Object naming is vital for the message transformation transactions to take place in a reasonable way. Each graph in a streams stack shares a common naming scheme with its neighbours; an object can be transformed into an identical copy (a "clone"), or can be split into a number of smaller objects, or can join with other objects to form a larger object.

If an object responds to some action by producing a collection of actions to be passed on to its clones in other domains, then it is the responsibility of the transformations to produce valid "addresses" for these messages — the object itself has no idea of what object-associations may be defined, so the only addresses it can provide are symbolic ones based on its own name ("all objects linked via transformations to me, object a"). A transformation must convert these symbolic addresses into "real" object references in each of the domains in question.

Of course, a transformation will generally do a lot more with an address than simply convert it and pass it on; the most useful kinds of transformation are those which selectively modify the message streams passing through them, essentially doing pattern-matching on their addresses as a key to what to change. The disadvantage of such a scheme is that the messages themselves have to be examinable, which makes functional specification of messages impossible.

In fact, this is not quite the case — it is possible to convert such functions in terms of combining them with existing functions in the appropriate domain-space. But this leads to a vast amount of traffic through the transformers; they must be converted back and forth every time such functions are activated, as they are not situated in a single domain. The addressing problem becomes even more complex in this case, so it would be reasonable to prohibit such things.
Chapter 6

Graph Domains

6.1 Structure

The structure for a graph-based interactive transformation system is built up by inserting the details of the "graph" system into the general "transformational" system. Thus, what results is a network of graph-domains, connected together by event-message transformations.

6.2 Attribute Control

An attribute of an object in a graph domain is controlled by the sending of event messages addressed to that object; this addressing is either directly, "by name" from another object in the same graph domain, or else is by translation from an address generated in another graph domain. Of course, the user-level interface itself has the ability to directly address objects in the user-level graph, by name (through actions such as pointing a mouse at them).
Once an event message has been successfully delivered to an object, the object then sub-addresses the message to whatever attributes it effects. Since attributes are the analogue of "methods", this corresponds to the activation of a method in response to an event. The activation of an attribute just causes its evaluation function to be called (or, in the case of meta-regenerative attributes, for all appropriate regenerations to be called all the way down to the zero-level — *i.e.*, the evaluation function itself).

In the case of the existing XGE implementation, an event message contains the name of the object to target on, the name of the attribute to alter (only one attribute per event message), and a description of how to alter it.

Possible alterations available include:

- **replacement** — a new evaluation function is installed. This is necessary if no provision for meta-regeneration is provided.

- **shifting** — XGE represents attribute specifications using a combination of functional terms and fixed terms. Thus it is possible to add\(^1\) an offset or increment to the existing value. This has the advantage that such alterations are *identifiable*; not being functional, modifications of this type can be compared. As a special case, shift alterations can be easily reversed, as their structure is immediately available for analysis.

\(^1\)The word "add" is perhaps misleading in this context, as value-shifting can also be used to subtract from a value (such as removing potential items from an unevaluated list).
Chapter 6. Graph Domains

- **touching** — such an event doesn’t change the attribute at all, but causes it to be evaluated nonetheless. This is useful for those attributes which correspond to inherently non-functional activities (such as drawing on the output display).

In addition, a few events exist which deal with objects as single entities, rather than at the attribute level. Examples of these are:

- **create** — create a new object, installing some initial values for a set of named attributes. In fact, this is redundant, as creation occurs automatically if an event message attempts to address a not-yet-existing object\(^2\). This greatly simplifies matters, as it is not necessary to ensure that an object has been explicitly created before acting on it.

- **delete** — destroy the object. Such an event must be broadcast throughout the domain, as other objects may need to respond to it (to allow them to clear out any references to the about-to-be-deleted object). Since this is a necessary occurrence, it must be done at the level of the event-manager for the domain rather than leaving the responsibility for re-transmission in the hands of the *delete* event-handler in the object in question itself. Once an object has been deleted, it must never be referenced again.

\(^2\)This is provided as an efficiency measure.
6.3 Concrete Attributes

Treating an attribute as a potential “method function” event handler, a graph object must provide all of the following:

- **appearance** information attributes, to implement the physical drawing of the object on an output display, and to permit the assignment and retrieval of the structure defining this appearance.

- **subobject** information, describing the collections of subobjects (of the same type) embedded within this object (and permitting the assignment and retrieval of this information).

- **linkage** information, describing the edge-structure of the graph, as seen from the vantage-point of this object. For a node or edge object, this will be just information describing the interconnecting port objects, whereas for a port object it describes the incident arc objects and the associated node object.

- **data** information, describing the data tagged to this object.

Of course, an actual implementation can handle these abstract requirements in any way it sees fit — the XGE implementation provides the following:

- **appearance**.

  Appearance is handled as a collection of independent attributes. Firstly, the size and position of the object is specified in relative terms to its “parent” object (the object that connects it to the main part of the graph object-tree as a subobject). Secondly, the “drawing” information is treated as simply a generic drawing operation to be called on the data attribute — this simplifies
Chapter 6. Graph Domains

matters a lot, but it does preclude an object from taking responsibility for the details of its own drawing; here is a place where a more object-oriented approach might seem more suitable.

- **subobjects.**
  Subobject information is simply a description of the subobjects.

- **linkage.**
  Linkage information is as expected — simply descriptions of the appropriate linked objects.

- **data.**
  Data information is handled in two ways. Firstly, data that corresponds to physical appearance is bundled up into a *shape* attribute. A shape is a hierarchical structure built up from basic diagram-drawing primitives. It provides methods for drawing lines of various kinds, text strings in various sizes and fonts, and finally new trees of graph objects. This shape information is expected to be interpreted as a "label" for the object; so it makes sense to allow labelling of an object with structures built out of these.

  Secondly, data which is "private" to the object is stored simply as a table of values. This is intended for attaching semantic meaning to objects, independently of the actual physical appearance. Of course, the appearance attributes might depend on these internal data structures, so it is possible to get them to show up in the display if really needed. The table of such data

---

3Of course, such an object-oriented approach for display handling is not inconsistent with the overall graph network representations used in XGE. A global display strategy was adopted for simplicity.
subattributes is currently implemented as a list of “name, value” pairs, where both the name and the value are simple text strings. This is obviously not particularly flexible, but is trivial to implement. It would certainly be useful to be able to associate data of any type whatsoever with an object.
Chapter 7

Implementation of Graph Domains

The following chapters describe a particular implementation of a graphical interface which is based on transformational graph domains, the XGE graph editor. XGE is written in the Standard ML functional programming language.

This chapter gives details of the XGE implementation of graph structure, based on the general design discussed in chapter 4.

7.1 Datatype Representation

In order to build up large structured networks of objects, a means of building references to objects must be developed.

First, it is useful to treat references as "pointers", so that an object may reference another object "by name" in an efficient way simply by its containing a copy of its defining reference\(^1\). This can lead to infinitely recursive type definitions.

There are several ways of overcoming this problem, of which the simplest are:

\(^1\)This is a "reference" in the ML sense; a ref corresponds to a "pointer" or "address".
• implement pointers, by using a pointer datatype in place of a simple reference, which allows the possibility of a null value.

• always use lists instead of simple values, so a null value is represented by an empty list.

The second of these seems clumsy, so the first option has been adopted in the XGE implementation. A generic datatype is used:

```haskell
datatype 'a pointer =
  ptr_null
  | ptr_ref of 'a ref;
```

A second problem is also caused by recursive type definitions — namely that only a datatype type may be recursive. Thus, implement all the major types are implemented as trivial datatypes of the form

```haskell
datatype thing = constructor of sometype;
```

This is somewhat ugly, as a simple use of sometype should have sufficed.

### 7.2 Design

#### 7.2.1 Descriptors

The fundamental means of referring to an object is via an identifier. An identifier is represented by a pointer to an entity of the appropriate type. The most commonly used is called simply Id and is defined simply as

```haskell
type Id = Object pointer;
```
this is used to refer to objects within a graph. Similarly, identifiers are available for referring to graphs, windows, etc.

7.2.2 Types

Attributes of objects are implemented using the ObjectField type — all attributes are of the same type, much simplifying the use of generic operations on attributes. As these attributes, corresponding to fixed interpretations, are provided only for optimisation, the limitations imposed by a common type are unimportant; these attributes are being used for particular fixed purposes, by definition.

A few other types are provided for specialized uses such as labelling the "type" of an object (as node, edge, port, etc).

7.2.3 Geometry Values

Two special types are used for describing positions and sizes. Each comes in two forms — a definite form describing a value, and a hazy form which allows for explicitly and implicitly undefined values.

The definite forms are given by:

\[
\begin{align*}
\text{type } \text{POSITION} &= \text{int} \times \text{int}; \\
\text{type } \text{SIZE} &= \text{int} \times \text{int}; \\
\end{align*}
\]

where each is simply a pair of integers. All dimensions are interpreted relative to a virtual 1000x1000 frame — for an object, this is the frame defined by the dimensions of its parent object.

The hazy forms are given by the following datatypes:
Chapter 7. Implementation of Graph Domains

```
datatype Position =
    pos_undefined
    | pos_default of POSITION
    | pos_value of POSITION;

datatype Size =
    size_undefined
    | size_default of SIZE
    | size_value of SIZE;
```

where the `pos_undefined` and `size_undefined` components are for explicitly undefined values of positions and sizes, the `pos_default` and `size_default` components are for implicitly defined (or undefined) values, and the `pos_value` and `size_value` components are for explicitly defined values.

### 7.3 Details

A graph is implemented as a network of updatable object-nodes. In Standard ML, this corresponds to lists of references to such things. The linkage information defining the structure of a graph is included as part of the attributes of the objects.

Concretely, an object in a graph is represented by the following type definition. The individual subtypes are described in full in the next section.

```
datatype Object = OBJ of ObjectRecord;
type ObjectRecord = {
    o_type : ObjectType,
    o_name : ObjectName,
}```
Chapter 7. Implementation of Graph Domains

```cpp
    o_context : GraphId,
    o_parent : Id,
    o_position : ObjectField,
    o_size : ObjectField,
    o_labels : ObjectField,
    o_subobjects : ObjectField,
    o_ports : ObjectField,
    o_picture : ObjectField,
    o_links : Id list
};
```

The individual fields are described in full as:

- **o_type : ObjectType.**
  An ObjectType is defined as

  ```cpp
datatype ObjectType =
    obj_node
  | obj_arc
  | obj_port
  | obj_label
  | obj_null;
```

and simply describes what type of object this is. The obj_null selector is used to refer to objects of "unknown" type, and is provided to allow completely 'empty' objects to be present in a graph, which can be filled out at a later stage.

- **o_name : ObjectName.**
  An ObjectName is defined as
Chapter 7. Implementation of Graph Domains

```haskell
datatype ObjectName =
  oname_atom of Id
| oname_filt of FilterId * ObjectName
| oname_filtgroup of FilterId * (ObjectName list)
```

where the various selectors correspond to:

- **oname_atom.**
  this is simply an identifier for this object (a self-reference, essentially). This is the underlying naming method, and is how all objects in basic (ie, internal) graphs are named.

- **oname_filt.**
  this is the naming method used for an object in a non-basic (ie, external) graph that has been altered from an object in a basic graph by some filter. The FilterId refers to the filter used, and the ObjectName refers to the object (in a basic graph) on which it is based. An object in an external graph that has not been modified by any filter (ie, it is simply a clone of an internal object) is named by an oname_atom constructor.

- **oname_filtgroup.**
  this is used to name a "virtual" object. These are produced in external graphs by filters, based on a collection of internal objects. The filter and the base collection are described by the FilterId and ObjectName list fields.

- **o_context : GraphId.**
  This refers to the graph to which the object belongs.

- **o_parent : Id.**
  This is the "parent" of the object — that which contains the object as a
subobject. The parent of a port object may be a node object at the point at which the port is connected to the rest of the graph, but otherwise the parent will be of the same type as the object itself. It is this which describes the structure of a hierarchical graph, together with its inverse, the subobject field.

- **o_position**: *ObjectField*.
  
  This describes the position of the object relative to its parent.

- **o_size**: *ObjectField*.
  
  This describes the size of the object relative to its parent.

- **o_labels**: *ObjectField*.
  
  This describes the label objects associated with this object (which define annotations on how the object is to be physically displayed).

- **o_subobjects**: *ObjectField*.
  
  This describes the list of subobjects of this object, to build up the hierarchical tree that is a graph.

- **o_ports**: *ObjectField*.
  
  This describes the port objects associated with this object which connect edges.

- **o_picture**: *ObjectField*.
  
  This describes a *picture* associated with this object, which is how the object is to be physically displayed. Every object contains just one such picture — additional ones can be associated with the object using *o_labels* labels.

- **o_links**: *Id list*.
  
  This describes the links associated with this object, other than those which
are explicitly described elsewhere. For an arc object, this is a list of ports (normally exactly two ports — any other number leads to a hypergraph). For a port object, this is a list of incident edges. For any other type of object, it has no meaning (and so will always be nil).

7.4 Attributes

Not all attributes are dealt with using the ObjectField type — this is unfortunate, but is probably inevitable given the clumsiness with which this type is defined. The major advantage of a common type, that it makes it easy to manipulate attributes uniformly, more than makes up for this.

An attribute which has type ObjectField is special in that it allows the following:

- Specification.

  The value of the attribute is determined by evaluating a specification function, passing the identifier for the object itself along with a list of other objects on which it has been stated to depend. This evaluation results in two values:

  o Attribute Value.

    This is the new actual value for the attribute

  o Dependency Value.

    This is a new list of dependencies — the list which is to be passed on subsequent evaluations of this specification.

The functional form of the specification serves two purposes. First, it allows the values to be cached — the computed attribute value can be used until it is no longer valid, due to a change in one of its dependencies, at which point
it can automatically be re-evaluated. Second, it provides a means by which specifications can be programmed in the state-machine sense, where, again, the changes in dependencies trigger off state transitions.

- **Automatic Dependency-Based Re-Evaluation.**
  Whenever an attribute of an object changes (for whatever reason), all attributes of other objects which explicitly depend on the changed attribute will be automatically re-evaluated, without any need for this to be handled explicitly. Only a single cycle of such re-evaluations takes place, so that a pair of objects with inter-dependencies will not cause infinite regression.

- **Data Hiding.**
  In principle, the kind of data being handled can be of any type, as it is only manipulated by functions provided with the specification itself. In practice, the XGE implementation only permits a fixed set of possible types.

An **ObjectField** has the following type:

```plaintext
type ObjectField = {
  ofield_spec : ObjectFieldFunction,
  ofield_value : ObjectFieldElement,
  ofield_depend : Dependency list,
  ofield_fuzz : Fuzz
};
```

where the **ObjectFieldElement** type is defined as

```plaintext
datatype ObjectFieldElement =
  el_null
  | el_position of Position
```
| el_size of Size
| el_labels of Id list
| el_subobjects of Id list
| el_ports of Id list
| el_picture of Viewer * LabelData
| el_userdata of Userdata
| el_locks of Locks;

where

- `el_null` is a null value, describing an undefined attribute,

- `el_position` describes the position of an object,

- `el_size` describes the size of an object,

- `el_labels` describes a set of labels associated with an object,

- `el_subobjects` describes a set of subobjects of the same type,

- `el_ports` describes a set of associated port objects,

- `el_picture` describes a picture associated with this object — the `Viewer` component describes view-based data (such as zooming, panning, etc) while the `LabelData` describes the format of the picture itself, and is described in full in the chapter detailing the implementation of the XGE windowing system.

- `el_userdata` describes user-defined data to be associated with the object — currently this is restricted simply to lists of named strings:

    ```
    type UserData = (string * string) list;
    ```
• el_locks allows certain attributes of certain objects to be “locked” against various actions (so that, for example, an object may be locked against size-change if its given size is critical in some way).

The individual components of an ObjectField are as follows:

• ofield_spec : ObjectFieldFunction.
  This is the actual specification function of type defined as a function taking an object identifier and a list of dependencies and returning a value and a new list of dependencies.

  type ObjectFieldFunction =
  
  (Id * Dependency list)

  ->

  (ObjectFieldElement * Dependency list);

• ofield_value : ObjectFieldElement.
  This is a cache of the current value (for efficiency purposes — in principle there is no reason why the value should not be re-evaluated from scratch whenever it is needed). This caching tends to enforce the requirement that the specification function depend only on its given parameters.

• ofield_depend : Dependency list.
  This is a list of dependencies for this attribute, defined by the type

  type Dependency = Id * DependencyField;
  datatype DependencyField =
    dep_null
    | dep_position
    | dep_size
Thus, a dependency consists of an object identifier (the object on which it depends), and an attribute identifier (to identify the individual attribute on which the dependency lies). Dependencies are thus implemented at the attribute-attribute level.

• \texttt{ofield\_fuzz : Fuzz.}

This is a means by which the result of the evaluation of a specification may be "offset" by some amount. This is useful when combining purely functional specifications with purely definitional ones. (Definitional ones come about naturally by interaction — it isn't too easy to define functional specifications at run-time, using a keyboard and mouse). Their existence also permits a crude form of analysis of the form of an attribute value, which would be impossible otherwise due to the inherent non-visibility of the internal structure of a function.

The \texttt{Fuzz} type is defined simply as a pair of value-lists — the first specifying items to "add" to the existing value, and the second specifying items to "take away". (Obviously this is not applicable to all attribute types.)

\begin{verbatim}
    type Fuzz = ObjectFieldElement * ObjectFieldElement;
\end{verbatim}

The eventual value of an attribute is the combination of the evaluated specification and the fuzz offsets.
7.5 Labels

All objects have a `picture` attribute. This describes how the object should be displayed. Only a single diagram may occur in each such picture, so more complicated ones may be described with a tree of `label` objects associated with the object, each containing a picture representing some aspect of the total display.

The `el_picture` attribute of the object itself should probably only be used for very fundamental picture components, such as a bordering frame.

A picture is described by a pair consisting of a `Viewer`, describing local distortions to be applied (such as zooming, panning, etc), and a `LabelData` describing the details of the picture.

The `LabelData` type is a datatype containing the following constructors:

```plaintext
LabelData =
  label_null
  | label_string of string
  | label_diagram of Diagram
  | label_icon of Icon
  | label_node of Id
  | label_text of (string * Font);
```

The various picture types are:

- `label_null`.
  
an empty picture — nothing.

- `label_string`.
  
a line of text to be displayed in a default font.
Chapter 7. Implementation of Graph Domains

- **label_diagram.**
  a diagram (collection of lines to be drawn).

- **label_icon.**
  an iconic picture or bitmap to be displayed.

- **label_node.**
  a graph to be drawn as a label.

- **label_text.**
  a line of text to be drawn in a specified font.

The **Font** type is an X primitive type, the **Icon** type is a hidden XGE type, and the **Diagram** type is an open XGE type, defined as:

```plaintext
type Diagram = {
  diag_lines : (POSITION list) list,
  diag_curves : (POSITION list) list
};
```

where the **diag_lines** component is a list of straight lines to be drawn, and the **diag_curves** component is a list of curved lines to be drawn by spline extrapolation.

Icons are manipulated by the two functions:

- **GetIcon(iconname:string, tiling:bool) : Icon.**
  This fetches an icon from a named file. The **tiling** parameter allows tiled icons to be constructed (for repeating patterns).

- **PutIcon(icon:Icon, filename:string) : unit.**
  This writes an icon into a file. It is not currently available — though this
is not too important, as there is no means for editing icons from within XGE anyway, so all icons must have originated in files to start with.

Fonts may be read in from files using the function

\[
\text{GetFont(fontname:string)} : \text{Font};
\]

Both fonts and icons may be named either by full pathnames of the files containing their definitions, or else simply by a name of the font or icon to be searched for in a standard place. Attempts to load a non-existent font or icon are not handled with much sophistication, so it is advisable to only use well-known ones.

### 7.6 Implementation Interface

The following function calls implement the graph-level interface to the system.

- **GetObject (id:Id) : ObjectRecord.**
  This returns all the information contained in the object referred to by the object identifier `id`.

- **GetObjectType (id:Id) : ObjectType.**
- **GetObjectName (id:Id) : ObjectName.**
- **GetObjectContext (id:Id) : GraphId.**
- **GetObjectParent (id:Id) : Id.**
  These extract the fields in question from the object referred to.

- **GetObjectPosition (id:Id) : Position.**
- **GetObjectSize (id:Id) : Size.**
- **GetObjectLabels (id:Id) : Id list.**
GetObjectSubobjects (id:Id) : Id list.
GetObjectPorts (id:Id) : Id list.
GetObjectLinks (id:Id) : Id list.
GetObjectPicture (id:Id) : ObjectFieldElement.
GetObjectLocks (id:Id) : Locks.
GetObjectUserdata (id:Id) : UserData.

• GetObjectAttribute (id:Id, f:DependencyField) : ObjectField.
  This returns the attribute-field of any named attribute of an object.

• GetObjectWindow (id:Id) : WindowdataId.
  This returns the identifier for the window in which an object is contained in
  a graph. If the object is in a base graph (ie, not in a window at all), then an
  exception will be raised.

• GetObjectInstantiation (id:Id) : bool.
  This returns the status of the “subobject instantiation” of the named object.
  If the subobjects are instantiated, then they actually exist; otherwise lazy
  evaluation is assumed, and they will not exist until explicitly asked for (with
  the SetObjectInstantiation call).

• GetObjectFuzz (id:Id, f:DependencyField) : Fuzz.
  This returns the “fuzz” component of the attribute named by the field f.

• GetObjectPositionFuzz (id:Id) : Position.
  GetObjectSizeFuzz (id:Id) : Size.
  GetObjectLabelsFuzz (id:Id) : Fuzz.
  GetObjectPortsFuzz (id:Id) : Fuzz.
  GetObjectSubobjectFuzz (id:Id) : Fuzz.
Chapter 7. Implementation of Graph Domains

GetObjectUserdataFuzz (id:Id) : Fuzz.
GetObjectLocksFuzz (id:Id) : Fuzz.
These return the "fuzz" value of the appropriate attribute of the named object. These are not strictly speaking necessary, as the more general GetObjectFuzz can be used to implement all of these. They are provided for ease of use.

- SetObjectAttributes (id:Id, attr:Attribute list) : unit.
This sets all the attributes given in the attr list in the named object id. An Attribute has type

    type Attribute = ObjectField * DependencyField;

consisting of a value (the ObjectField) and a descriptor for the attribute to be set (the DependencyField).

- SetObjectPosition (id:Id, spec:ObjectField) : unit.
SetObjectSize (id:Id, spec:ObjectField) : unit.
SetObjectLocks (id:Id, spec:ObjectField) : unit.
SetObjectUserdata (id:Id, spec:ObjectField) : unit.
SetObjectPorts (id:Id, spec:ObjectField) : unit.
SetObjectLabels (id:Id, spec:ObjectField) : unit.
SetObjectSubobjects (id:Id, spec:ObjectField) : unit.
SetObjectPicture (id:Id, spec:ObjectField) : unit.
These set explicitly-named attributes.

- SetObjectPictureConstant (id:Id, view:Viewer, ld:LabelData) : unit.
This sets a "constant" picture attribute. Since it is likely that most pictures will be constant rather than necessarily functional, this is provided as a simplified interface.
Chapter 7. Implementation of Graph Domains

- SetObjectFuzz (id:Id, dfl:DFL list) : unit.
  This alters the "fuzz" for a list of attributes — the new fuzz values will be the old ones combined with the modifications specified. The DFL type is defined as

  type DFL = DependencyField * Fuzz;

- SetAbsoluteObjectFuzz (id:Id, dfo:DFO list) : unit.
  This sets the fuzz values absolutely — the new fuzz values are exactly those defined here. The DFO type is defined as

  type DFO = DependencyField * ObjectFieldElement;

  Note that the DFL and DFO types are not globally available as type names — I just use them here for convenience.

- NewObject (typ:ObjectType, gr:GraphId, attr:Attribute list) : Id.
  This is used to create a new object. It is created of type typ in the graph gt. Its initial attribute values are set from the attr list.

- DestroyObject (id:Id) : unit.
  This destroys an existing object. There must be no references to this object in existence at the time — any other objects that still refer to this one will themselves be destroyed too. (This is an easy way to destroy an entire graph, by mistake.)

- InstantiateObject (id:Id) : unit.
  This sets an object to have instantiated subobjects — so that real rather than lazy evaluation will be used for them. Lazy evaluation is provided for two
reasons: first, for efficiency, so that time is not wasted on evaluating objects that are never used; second, to provide a facility for potentially infinitely deep subgraphs.

- **UninstantiateObject** (id:Id, deflect:Deflection) : unit.

  This causes the subobjects of a given object to become lazy, in that they will never be created until explicitly required. The `deflect` function is used to temporarily cause any edges that currently pass to these subobjects to be deflected to this object itself (as they cannot link to the non-instantiated subobjects). The type `Deflection` is defined as

  ```haskell
  type Deflection = ObjectRecord -> (Attribute list);
  ```

  and is used to give initial attribute values to the temporary ports that will be set up (one for each edge deflected). Any attributes not set in this way will be given some default value.

- **MakeSpecification** (ofe:ObjectFieldElement) : ObjectField.

  This can be used to produce "constant" attribute specifications. The resulting `ObjectField` has as its `ofield_spec` component a constant function which will always return `ofe`.

Two object values are available — **VOID** and **NULL**. **VOID** is an object with all its components unset, whereas **NULL** is a null object pointer. For consistency, **NULL_OBJECT** is a synonym for **NULL**.
Chapter 8

Implementation of Graph Filters

8.1 Introduction

A graph filter is a means of implementing a transformation between a pair of graphs. Any modification made to objects in one of the graphs is passed through the filter to form a modification which must be applied to the other graph — this application will of course occur automatically.

The term “modification” is used to refer to the changes which must be made to a graph in order to produce an associated filtered graph. Changes made within a single graph, as described earlier, act orthogonally to this. The model that is being built up is one in which a number of graphs are connected vertically by filter streams, in such a way that if an upper-most or bottom-most graph is altered in any way, then a set of modifications is generated and passed down (or up) the filter stream to produce compatible alterations to all the graphs in the stack.

For this purpose, modifications are generated automatically whenever a particular graph is altered. The generation and propagation of modifications through the
filter streams are under the control of the underlying system, whereas the ways in
which individual filters can react in response to these modifications is intended to
be programmable, by means of filter module specifications.

First I will describe the format of the fundamental message-type which makes
up such modifications. Application of a modification is simply a list of these funda-
damental messages being targeted on a graph.

8.2 Message Formats

Modification of a graph is implemented as a list of type Modification, where

\[
\text{type Modification} = (\text{ObjectRecord list}) * (\text{FilterModType list});
\]

ie, each item in a modification list consists of a list of objects to be changed and a
list of the changes to be made.

The FilterModType type is a disjoint-union of possible modifications, defined
as:

\[
\text{datatype FilterModType} = \\
\text{fmod_null} \\
| \text{fmod_func of DependencyField * ObjectFieldFunction} \\
| \text{fmod_depend of DependencyField * (Dependency list)} \\
| \text{fmod_fuzz of DependencyField * Fuzz} \\
| \text{fmod_attr of DependencyField * ObjectField} \\
| \text{fmod_delete} \\
| \text{fmod_create};
\]

The kinds of modification are:
• \textit{fmod\_null}.

This is the \textit{null} modification, that does nothing at all.

• \textit{fmod\_func}.

This changes the functional definition of an attribute, by giving it a new value. The attribute to be changed and the new functional specification value are passed as parameters.

• \textit{fmod\_depend}.

This changes the dependency relations associated with an attribute. The name of the attribute to be changed and the new dependency list are passed as parameters.

• \textit{fmod\_fuzz}.

This changes the \textit{fuzz} offset associated with a particular attribute. The name of the attribute and the new fuzz value are passed as parameters.

• \textit{fmod\_attr}.

This changes a complete attribute value — the functional specification, the dependency list and the \textit{fuzz} value.

• \textit{fmod\_delete}.

This is a request that the object being addressed be deleted (i.e., removed totally from the graph domain in question). As it is to be totally destroyed — not just unlinked from the graph structure — there must be no references to it from other objects, otherwise they too will be deleted. If such objects are required to remain (with these references transferred to some other object, presumably), then they must be reconnected elsewhere before the deletion message is sent.
Since objects are implicitly created when addressed, the deletion of a non-existent object will simply cause it to be created and then immediately deleted, without causing any harm.

- `fnod_create`.
  This creates a new object, with its attributes initially unset. A new object will be automatically created if a non-existent object is addressed, so this is not specifically required. It is included for symmetry.

### 8.3 Filter Types

A filter can be thought of as a "uniform" alteration of a graph, to create another one — this is reasonably close, in fact, as an alteration of the instructions which go to make up a graph is equivalent to an alteration of that graph itself. This use of the word *uniform* is intended to force these alterations to be local relative to the overall structure of the graph — an alteration can only have effect in a local region around the objects it addresses.

To simplify matters, I only permit a very limited number of operations to be implemented in terms of filters. These are:

- **Distortion**.
  A filter is permitted to distort the attributes of the objects it addresses. This obviously will leave the overall structure of the graph mostly unchanged. (In fact this is not quite true, as some attributes contribute to this structure — subobjects, links, etc — but what is significant is that no important restructuring can occur, as the object being addressed must maintain its place.)
• **Deletion.**
  A filter can delete an object — this causes the addressed object to simply not appear at all in the filtered graph. Thus, objects can be “virtually hidden” using this.

• **Combination.**
  A filter may “virtually combine” several objects in the original graph to create a virtual object which contains images of all its constituents as subobjects.

This has two uses:

- Firstly, it may be used to “hide detail”. An entire subgraph may be “combined” and its constituent parts then “deleted” (ie, hidden), to cause it to appear as a single object.

- Secondly, it may be used to enable the simultaneous manipulation of several objects. This may be done by combining them to form a virtual object and then performing any manipulation on this. Any changes will be back-filtered and performed individually on each of the original objects in the pre-filtered graph.

The type of a filter is thus determined by the following datatype. The *distortion* and *deletion* kinds are simply labels describing the type, whereas a *combination* type must contain additional information describing how to form the new object out of the original ones.

```haskell
datatype FilterType =
  filt_distort
  | filt_delete
  | filt_combine of {
    fcomb_distort : (FilterMatch * FilterActionModule)
```
8.4 Addressing

A filter action operates on all objects whose name matches the "address" associated with it. It would be convenient to have full set-operations defined for building up such addresses, so that matching would be a simple pattern-matching operation. However, I only provide the following, without any "higher-order" matching.

```
datatype FilterMatch =
  match_targetlist of Id list
  | match_union of FilterMatch * FilterMatch
  | match_intersection of FilterMatch * FilterMatch
  | match_complement of FilterMatch;
```

Address matching on these has the obvious semantics, with the base level being that of `match_targetlist` — an object matches a target-list if it is contained in that list.

The lack of higher-order matching means that complicated address specifications, such as "all ports connected to object x", cannot be handled. This could be easily fixed by adding a case for a

```
match_func of (Id -> bool)
```

constructor, with the semantics that an object name matches such an address if the function returns `true` when applied to it.
8.5 Filter Action

A filter operates by selecting those objects which match various different filter addressing patterns, and applying the appropriate action routines to these. An action routine is simply:

```haskell
type FilterAction = ObjectRecord -> FilterModType;
```

being a function which returns an alteration specification when presented with an object record. This alteration is then applied to the image of the object in question, in the target graph.

Since a filter is a two-way entity, it must be possible to filter in both directions. Ideally, it should only be necessary to define the filter in one direction, from which its inverse could be automatically determined, but this is not possible in general, given that pure (non-symbolic) functional specifications are being used.

Thus, a pair of functions must be given for each action — the "forward" action and the "backward" action. These must be inverses of each other.\(^1\) So, actions are paired into:

```haskell
    type FilterActionModule = (FilterAction list) *
                             (FilterAction list);
```

---

\(^1\) This is so that combinations of filterings and reverse-filterings can cancel out; if this does not happen, then the graphs will gradually become more and more inconsistent with each other.
A particular form of action is that needed to “virtually combine” objects. For this, the alterations are computed from a list of objects rather than a single one; this list is, of course, simply the list of objects being so combined. The following type is used to represent this:

```haskell
type FilterCombineAction = (ObjectRecord list) -> FilterModType;
```

### 8.6 Filtering Summary

Filtering is thus controlled by the following types. A `FilterId` type is simply a pointer to a `Filter`, which is a datatype for a `FilterRecord`. A `FilterRecord` is just a “stacking” object which contains lists of objects of type `FilterConverter`. In fact, the association is more general than a stack — a filter record also contains a list of associated extra filters, which are to be applied in turn after the converter list is exhausted — producing a structure whereby arbitrary sub-filters can be inserted or deleted at any point.

```haskell
type FilterConverter = {
    conv_type : FilterType,
    conv_match : FilterMatch,
    conv_action : FilterActionModule
};

type FilterRecord = {
    filt_conv : FilterConverter list,
    filt_assocfilt : FilterId list
};
```
type FilterId = FilterRecord pointer;

8.7 Filter Action (continued)

A filter operates on an object (actually, on a list of attribute definitions for that object), by trying each converter in the filt_conv list. If a converter matches, by its conv_match address pattern-matching to the name of the object in question, then the associated action conv_action occurs, and that converter-list is aborted. Control then continues with the next stack of filters — the filt_assocfilt list.

If a converter matches an object, the action to be done will depend on the type of the filter — the conv_type field. A distort type is the most straightforward — the actions in the conv_action field operate on the object to produce modifiers to be applied to the object itself. If the filtering is “forward”, then the first list in the conv_action pair is used, otherwise the second list.

A delete type filter simply produces a modifier which will delete the object.

A combine type filter is the most complex. The fcomb_distort field is used to distort some of the constituent objects in particular ways. The fcomb_newobj field is used to produce a new object, and the general filt_action field is then used to make general distortions to all those objects which were not altered by individual fcomb_distort actions. This is perhaps excessively general, but it does allow for very tight control of exactly what happens to every object, while minimizing the details of specification required.
8.8 Filter Stack Operations

As a filter is somewhat like a streams stack, it is possible to push and pop filter modules associated with any given filter identifier. The implementation actually provides a two-dimensional tree-like structure, rather than a stack, which generalises stack-alteration functions — it is easy to determine exactly what modules are effected by a stack-alteration, whereas a linear stack would require complete re-evaluation of everything whenever any single module was changed. (In fact, this is not quite true; it would be possible in most cases to isolate the effects and thus optimize the re-evaluation, but it would be a lot more complicated to implement.)

The following stacking operations are available:

- **NewFilter (conv:FilterConverter list) : FilterId.**
  
  This constructs a filter, with the converter list conv set in it.

- **DestroyFilter (fid:FilterId) : unit.**
  
  This destroys a filter, if it is no longer required. It must not be part of an established filter-module stack at the time.

- **GetFilter (fid:FilterId) : FilterRecord.**
  
  This extracts the details of the structure of a filter.

- **GetFilterConv (fid:FilterId) : FilterConverter list.**
  
  **GetFilterAssoc (fid:FilterId) : FilterId list.**
  
  These extract the converter list and associated-filters list respectively.

- **SetFilterConv (fid:FilterId, conv:FilterConverter list) : unit.**
  
  This assigns a converter list to a filter.
Chapter 8. Implementation of Graph Filters

- **SetFilterAssoc** \((f: \text{FilterId}, \text{assoc:FilterId list}) : \text{unit}\).  
  This sets a list of associated (stacked) filters.

- **AddFilterAssoc** \((f: \text{FilterId}, \text{assoc:FilterId list}) : \text{unit}\).  
  **RemoveFilterAssoc** \((f: \text{FilterId}, \text{assoc:FilterId list}) : \text{unit}\).  
  These add and remove to/from an existing set of associated filters in a named filter.

A value **NULL_FILTER**, of type **FilterId**, is available for use as a null filter descriptor.
Chapter 9

Implementation of Graph Displays

9.1 Purpose

This chapter deals with the issues involved in producing a map of a graph, as a physical display. This amounts to imposing a filter stream on the graph, providing a physical window, and then inserting a viewer to allow the top of the filter stack to be represented visually within that window.

The filter stack can then be used to produce virtual graph alterations — changes which are visible in the display, but do not correspond to any physical changes within the base graph itself.
9.2 Graph Structure

A \textit{graph} is an entity containing a description of a network of nodes, edges and ports. The graphs in questions are built up hierarchically, and thus a graph can be represented as a tree of objects. Each level of the tree corresponds to a particular 'depth' of hierarchy.

Since it is reasonable for a graph to contain unconnected objects (\textit{i.e.}, objects that have been created but not connected to the main structure, or transient objects which are in the process of being re-connected elsewhere in the network). Thus, a graph must contain both a description of the root of the tree and also a list of all objects in the graph (whether reachable from the root or not).

A third component describes the windowing associated with the graph. A graph is only visible by virtue of a mapping into a window. In fact, a window itself will contain a graph which is the result of this mapping, so this window-mapping description will depend on whether the graph in question is an internal graph or an external (window) graph. An internal graph will contain a list of references to the windows in which it is to be mapped, and an external graph will contain a single reference to the window in which it is embedded.

Thus, the following types are used:

```haskell
datatype GraphWindowData =
  grwin_win of WindowDataId
| grwin_am of WindowDataId list;

type GraphRecord = {
  gr_objects : Id list,
  ...
};
```
Chapter 9. Implementation of Graph Displays

```haskell
gr_root    : Id,
gr_window  : GraphWindowdata
```

```haskell
type Graph = GR of GraphRecord;
type GraphId = Graph pointer;
```

The fields of the GraphRecord are:

- **gr_objects**.
  This is a list containing all the objects in the graph. This list will always contain at least a single object — the root object itself.

- **gr_root**.
  This is a reference to the root object of the graph, of type `obj_node`. All other objects in the connected (i.e., visible) graph can be reached from the root via its various subobjects, ports, labels, etc.

- **gr_window**.
  This describes the windowing aspects of the graph. The type of the graph (internal or external) can be deduced from the value of this field; an internal graph will have a `grwin_aim` constructor, giving the list of windows in which this graph is mapped, whereas an external graph will have a `grwin_win` constructor giving the window where this graph is located.
Chapter 9. Implementation of Graph Displays

9.3 Window Structure

As described above, a window is simply a visual external map of a graph. It must describe how the external graph embedded in this window is derived from an internal graph, via a filter mapping. It must also describe the physical aspects of the window, such as its size, position, etc.

A window is described using the WindowdataRecord type, as given below.

```haskell
type WindowdataRecord = {
    win_basegraph : GraphId,
    win_filter : FilterId,
    win_wingraph : GraphId,
    win_view : Viewer,
    win_X : Window,
    win_size : SIZE
};

type Windowdata = WIN of WindowdataRecord;
type WindowdataId = Windowdata pointer;
```

The fields of the WindowdataRecord type are as follows:

- **win_basegraph**.
  This is a reference to the base (internal) graph from which the local window graph is derived.

- **win_filter**.
  This is a reference to the filter which is used to map the internal basegraph to the local window graph.
Chapter 9. Implementation of Graph Displays

- *win_wingraph.*
  This is a reference to the local (external) graph in this window. In effect, it is simply the value of the base graph when passed through the filter.

- *win_view.*
  This describes local viewing information associated with the window. This describes zooming and panning of the graph within the window. The following section describes viewing more completely.

- *win_X.*
  This describes the X interface to this window. The type *Window* is provided by the X system interface to *Standard ML*.

- *win_size.*
  This gives the physical size of the window, in terms of the fundamental X-based pixel units associated with the display. In fact, this information could have been left out, as it can be extracted from the *win_X* field, but it is provided for efficiency.

### 9.4 Views

A *view* in a window describes the virtual view of a graph (or, more accurately, of an object). Each label object in a graph may be provided with its own viewer, and the window as a whole provides an "initial" viewer. A subobject inherits the view-attributes of its parent, unless it decides to explicitly over-rule these.

A view is described by the following type:

```plaintext
type Viewer = {
```
view_id : Id,
view_size : Size,
view_position : Position,
view_colourmap : Colourmap

The colouring information is provided to allow for inverse-video viewing of objects. An enhanced implementation could provide full colour control.

datatype Colours =
  COL_BLACK
  | COL_WHITE
  | COL_NORMAL
  | COL_REVERSE;

type Colourmap = {
  colour_foreground : Colours,
  colour_background : Colours
};

The fields of a Viewer are as follows:

• view_id.
  This is an object which the graph-view is to be zoomed or panned to. If it is null, then the entire graph will be available, whereas otherwise only those objects “deeper than” the view_id will be visible.

• view_size.
  view_position.
  These describe what areas of the view_id object are to be visible. The window
will contain from the top-left view.position reaching across and down for view.size. Thus, a view.size of 1000x1000 is a full panning, from the given position to the bottom-right of the view.id. These position/size fields can be best considered as defining a chunk of the object which is to fill the window — all parts of the objects outside this chunk will not be visible.

- view.colourmap.

This describes the foreground and background colour of the object. The foreground colour is the colour in which the contents of the object is drawn, and the background colour is the colour of the “canvas” that they are to be drawn on.

Obviously, setting the foreground and background to be the same colour will result in the contents being indistinguishable from the background canvas.

Four different Colours are provided — COL_BLACK and COL_WHITE are absolute colours, whereas COL_NORMAL and COL_REVERSE are relative colours. An absolute colour is simply the colour as named, while a relative colour is evaluated in terms of its environment (ie, the current inherited view).

A window viewer must obviously provide absolute colourings, as there is no inherited view to fall back on. Viewers associated with individual objects of a graph may specify relative colourings, so that COL_NORMAL means “the same colour as my parent” and COL_REVERSE means “inverse-video relative to my parent”.

9.5 Graph and Window Operations

9.5.1 Graph Operations

- **NewGraph ( ) : GraphId.**
  This creates a new graph, returning a descriptor for it. It is initially an empty graph — the only object contained within it is its root.

- **DestroyGraph (gid:GraphId) : unit.**
  This destroys an existing graph. It must have been previously isolated (i.e., any window-mappings removed).

- **GetGraph (gid:GraphId) : GraphRecord.**
  This extracts the record information out of a graph descriptor.

- **GetGraphObjects (gid:GraphId) : Id list.**
  **GetGraphRoot (gid:GraphId) : Id.**
  **GetGraphWindow (gid:GraphId) : GraphWindowData.**
  These extract the individual gr_objects, gr_root and gr_window fields from a graph object.

- **TransferGraphObjects (fromgraph:GraphId, tograph:GraphId) : unit.**
  This transfers all objects from one graph to another. The root of the fromgraph is left unchanged, so that fromgraph is left as an empty graph. The transferred objects will not be connected to the main graph tree in the tograph, though any connectivity within themselves will remain.

- **TransferGraphObjectsDangerously (head:Id, to:GraphId) : unit.**
This is a slightly more general, but also much more dangerous version of \texttt{TransferGraphObjects}. All objects from the head downwards are transferred, thus allowing the transfer of a subgraph from one graph to another. There are several points of danger in calling this, so its use is discouraged:

- **connectivity.**
  There must not be any connectivity between the objects being transferred and the objects (if any) remaining.\footnote{\textit{ie}, the objects to be transferred must constitute a closed subgraph.} If there is, havoc will ensue.

- **instantiation.**
  The \textit{connectivity} restriction also applies to non-instantiated objects — any object which could potentially be instantiated from the transferred objects must also be transferred, otherwise it will be later instantiated in the new graph while still remaining positioned in the old graph.

- **roots.**
  The head object must \textit{not} be the root of its graph, as it is to be transferred too — if it were a root object, then the old graph would be left without a root, which is forbidden.

As the objects can be moved all in a single transaction, this is obviously a much more efficient way of transferring full subgraphs between graphs, provided these conditions are met.

An object \texttt{NULL\_GRAPH} is provided, as the null graph descriptor.
9.5.2 Window Operations

- **NewWindow** (pos:Position, siz:Siz, map:Colourmap) : WindowdataId.

  This creates a window, at the given position and of the given size. Any undefined values for position/size will be resolved by an interactive placement, as with most standard X applications. The Colourmap parameter describes the initial colouring of the viewer for the window. This initial viewer will of course provide no zooming, as this can only apply once a graph has been mapped in.

- **DestroyWindow** (win:WindowdataId) : unit.

  This destroys an existing window. The window should not be mapped into when this is called.

- **AimWindow** (win:WindowdataId, gr:GraphId, filt:FilterId) : unit.

  This maps an internal graph gr into the window win, using the filter filt. It is this which finally allows a graph to be subject to interaction.

- **UnAimWindow** (win:WindowdataId) : unit.

  This unmaps a window from whatever graph it is currently mapped to. This should be called before a window is destroyed with DestroyWindow.

- **GetWindow** (win:WindowdataId) : WindowdataRecord.

  This extracts the full record of information out of a window descriptor.

- **GetWindowBasegraph** (win:WindowdataId) : GraphId.

- **GetWindowFilter** (win:WindowdataId) : FilterId.

- **GetWindowView** (win:WindowdataId) : Viewer.
GetWindowWingraph (win:WindowdataId) : GraphId.
These extract individual fields from the record associated with a window descriptor.

- SetWindowView (win:WindowdataId, view:Viewer) : unit.
This sets a new viewer component in a window.

An object NULL_WINDOW is provided, as the null window descriptor.

9.5.3 Viewer Operation

- MakeViewer (reverse:bool) : Viewer.
This constructs a simple viewer, with no zooming or panning, and with either normal or inverse-video colouring, depending on the reverse parameter — a false value results in a normal viewer, while a true value will be inverse-video.
Chapter 10

Implementation of an Interactive System

10.1 Control Messages

The states of graphs associated with the system evolve by means of reaction to control-messages. These messages are produced either by an application program of some kind — the application to which the graph-system is to be an interface — or by user intervention.

The major cause of message production is intended to be the latter — user controlled interaction — with the application program merely reacting to this. The reactions may conceivably be long-term, such as for animation, but typically the majority of "wait state" time will be spent waiting for input from the user.
10.2 Production of User-Level Control Messages

The only real way in which a user may provide input is through a keyboard and mouse. Other higher-level systems (such as menus) are built up from these.

At the lowest level, it is possible to associate actions with keyboard and mouse events. At a higher level, a menu and object selection scheme permits more complex actions.

10.2.1 Input Events

In order to bind actions to events, these events must be namable. The following types exist for the naming of keyboard and mouse-button events.

```haskell
datatype MouseStatus =
    mouse_none
  | mouse_left
  | mouse_centre
  | mouse_right;

datatype MetaContext =
    meta_none
  | meta_shift
  | meta_control
  | meta_shiftlock
  | meta_meta;

datatype KeyStatus =
```

key_character of string * (MetaContext list)
| key_mouse of MouseStatus * (MetaContext list)
| key_none;

The MouseStatus and string components of a KeyStatus are for naming the type of event (the pressing of a particular key or button), while the MetaContext type is for describing additional context — such as the simultaneous pressing of shift or meta, etc.

The “null event” can be specially named by key_none.

10.2.2 Menu Bindings

A menu is simply a table of description/action pairs. The description part is the label for a particular menu-selection, whereas the action part is a function to be executed if that menu element is selected. This is implemented using the following type definitions:

```haskell
type MenuElement = {  
  menu_prompt : string,  
  menu_func  : KeyStatus -> int
};
type Menu = MenuElement list;
```

Note that the action-function must return an int. This is so that it is possible to determine afterwards which element was selected. Note that all user-level actions must return positive return values, as negative ones are reserved for use by the interaction system itself.

In the present implementation, the prompt field of a menu element is simply a string, describing the action. It would be reasonable to extend this to more general
type. In fact, this could be done by re-implementing the menu-driver entirely in terms of graphical objects — a menu would then be simply a graph containing various objects, and the associated driver routines would simply execute actions in response to mouse-clickings on these objects.

When a menu is activated, the system must be passed a full menu description (of type \texttt{Menu}), and also a specification of where to place the menu. This placement may be specified as relative to the current mouse position, to a named graph window, or to the full screen display itself:

\begin{verbatim}
datatype Placement =
  place_mouse
  | place_root
  | place_window of Windowatald;
\end{verbatim}

\subsection*{10.2.3 Key-Event Bindings}

Keys can be bound to "actions" (so that the appropriate action is executed on receipt of that keyboard event), using the following type:

\begin{verbatim}
type KeyElement = {
  key_key    : KeyStatus,
  key_action : MenuElement,
  key_ubiq   : bool
};
\end{verbatim}

The \texttt{key_ubiq} field is to specify whether this key-binding is to be ubiquitous — ie, whether or not the binding should hold even during menu-selection. In this way, mouse-buttons (or even normal keyboard keys) can be bound to particular actions when pressed on an object, without disturbing the selection of menu actions.
The key_action field of a key binding is a single MenuElement. Only its action-function is used — the prompt field is ignored. A better use of the prompt field would probably have been to display its value in a special window while the action is being executed, but the current implementation does not provide this.

### 10.3 Low-Level Interaction Operations

The following operations are available for menus and key-bindings.

- **CreateMenu** (place: Placement, pos: POSITION, menu: Menu) : MenuId.
  
  This function creates a menu as described by the menu parameter. Its physical position is described by the place and pos parameters.
  
  The nature of the return type MenuId is not relevant at the user-level; in fact, it is currently implemented as a unit ref, as its value has no inherent meaning, but is intended merely as an identifier for comparisons.
  
  The menu is of a permanent nature — it will remain in place, receptive to item-selection, until it is explicitly removed.

- **DestroyMenu** (menu: MenuId) : unit.
  
  This removes a menu created by CreateMenu.

- **PopupMenu** (place: Placement, pos: POSITION, menu: Menu) : int.
  
  This produces a pop-up menu, and expects a selection to be made from it immediately. If it was produced from an action bound to a mouse-button event, then selection will be made by releasing this button, otherwise selection will be made by the pressing of any mouse button.
Chapter 10. Implementation of an Interactive System

The int return value is the value returned by the action function in the menu element which is actually selected.

The menu will vanish as soon as a selection has been made and the associated action executed.

- SetKeyaction (kel:KeyElement) : unit.
  
  GetKeyaction (kst:KeyStatus) : KeyElement.
  
  These assign and interrogate a key-binding.

- GetKeyActions () : KeyStatus list.
  
  This produces a list of all key-events to which an action has been bound. They can then be individually interrogated using GetKeyaction to examine the details.

- UnsetKeyaction (k:KeyStatus) : unit.
  
  This removes an action-binding from a key event, leaving it unassigned.

- GrabMouse (cursor:string) : unit.
  
  To use mouse buttons which have been already grabbed by a window manager, or to provide input events to the system while the mouse is outside all display windows explicitly owned by the system itself, then mouse must be grabbed in this way for exclusive use by the graph system only.

  The cursor parameter describes what kind of mouse-cursor should be displayed while the mouse is grabbed in this way — it is the name of one of the cursor-fonts locally available. A null cursor parameter will un-grab the mouse.

  There should be no real need for this, for most applications. The most common reason for needing this is to allow use of mouse buttons which have already been grabbed by some other utility (such as a window-manager).
10.4 High-Level Interaction Operations

The following functions control interaction at a high level, in terms of influencing “flow of control”.

- **Pause()** : int.
  
  This is the “main loop” interaction function. It waits for input of some kind, and executes the appropriate actions, based on whatever menu and keyboard bindings are in force.

  Since the actions executed by these can in turn produce new menu and keyboard bindings, no more subtle control need be provided by the system itself.

- **Interaction()** : unit.
  
  This is an even higher-level function to control loops of interaction. It continually `Pause`'s until the `Terminate` action is executed.

- **Terminate()** : unit.
  
  This causes any pending `Interaction` call to terminate immediately, by raising an exception.

10.5 Object-Based Interaction

The following functions exist in order to be able to directly deal with objects at the interaction level. Essentially this is just a means by which objects may be referred to by pointing at them with the mouse.

All these functions should only be used in the evaluation of key-bindings or menu options.
• \texttt{mf\_position (valid:(Id->bool) list)}
  :
  \texttt{Id \times POSITION \times KeyStatus.}

This function takes a list containing zero, one or two "validation" functions, and allows the user to select an object using the mouse. Only those objects which the first validation function returns \texttt{true} on may be selected — attempting to select a non-valid object will cause an outward search from that object down to the graph root, looking for the closest encompassing valid object. If none is found, then no selection may be made at that point.

As the mouse is moved from one object to another, the first validation function is applied to the object. When the mouse leaves an object which is considered valid, then the second function in the list is evaluated on that object (this is to undo any temporary side-effects that the first function may have caused in that object, such as highlighting 11.2.3).

Eventually, a button (keyboard or mouse) will be pressed while the mouse cursor is in a valid object. There had better be a valid object, otherwise \texttt{mf\_position} will never be able to return. At this point, a descriptor for the object being selected, the position of the mouse relative to this object, and the keystatus describing what key made the selection, are returned.

There are two special features associated with the validation function list:

1. If the list is empty, then all selections are considered valid apart from selection of \texttt{arc} objects. An arc can only be selected if this has been explicitly enabled. The reason for this design decision is that arcs tend to swamp a graph, making selection of more interesting objects very difficult.

2. The first of the validation functions may be called multiple times before the "undo" second function is called. Thus, it should never cause any
“toggling” side-effects. Note also that no side-effects should occur to modify objects which are considered as non-valid for this selection.

- **mf_object** (valid:(Id->bool) list) : Id * KeyStatus.
  This is like **mf_position**, except that no position-information is returned. It is more useful when all that is needed is the selection of an object without reference to the details of whereabouts in this object the selection was made.

- **mf_unclickmouse** (valid:(Id->bool) list) :
  (Id*POSITION) * (Id*POSITION) * KeyStatus.
  This function may be used in the definition of mouse-button key bindings. All it does is wait for a mouse-button to be clicked or unclicked, then it returns the original object/position of the mouse before **mf_unclickmouse** was called, the object/position that the mouse is at when the button is clicked or unclicked, and the key status describing what button was involved.

  If this is used in a mouse-button key-bind definition, then it will be called up when a button is already down (the button that caused this key-binding to be activated), so it will wait for that button to be released. If used in some other context, when no button is down, then it will wait for a button to be pressed.

  Normally, button release events are ignored by the system, except when the function **mf_unclickmouse** is active.

- **mf_mouse** () : Id * POSITION.
  This returns the current object/position of the mouse. If the mouse is not in any object (an unlikely occurrence, since graph roots are themselves objects), then a NULL object identifier is returned.
Chapter 10. Implementation of an Interactive System

• **mf_buttons** *(place: Placement, pos: Position, menu: Menu) : unit.*

  This produces a temporary menu of buttons that may be used in conjunction with the other **mf_*** functions.

  The buttons will last for the duration of the current action (whatever action it was that caused these **mf_*** functions to be called up), and then they will automatically vanish.

  Whenever a **mf_position**, **mf_object**, **mf_unclickmouse** or **mf_mouse** function is called, a button from a buttons-menu may be selected instead of an object. This can be detected because the function in question will return a **NULL** object, and will return the return-value of the button-selection in both components of the mouse-position.

  **Mf_buttons** is a useful means for allowing an arbitrary collection of objects to be selected — the end of the collection can be marked by clicking on a special “end-of-collection” button.

• **mf_mousepath** *(draw: bool) : Id * (POSITION list).*

  This keeps track of the mouse from its present position up to until a button is pressed. At that point, the object in which the mouse was moving, and the path of positions it moved through, are returned.

  The path must not cross an object boundary — the positions are all relative to a single object. If the path does go outside the object (or if the mouse cursor is not in an object to start with), then a **NULL** identifier is returned, with an empty list.

  The **draw** parameter allows the optional drawing of a thin line as the mouse is moved. This line will be removed when the button is pressed, so it must be drawn properly again if a permanent record is to be kept.
The purpose of `mf_mousepath` is for user-definable drawings to be produced for labels on objects.

- `mf_string (prompt:string, raw:bool) : string.`
  This allows input to be read from the keyboard. The `raw` parameter chooses between raw (single keystroke) and cooked (a line of text) input. In cooked mode, the line is terminated by either a newline character or a mouse-click. Neither is returned in the resultant string.
  
The prompt string is currently ignored. In a fuller implementation, a cooked-mode read would pop up an editing window containing the prompt, and a line would be read with the use of text-editing.
Chapter 11

Appendix A: Examples and Applications using XGE

11.1 Summary

This section is not a user's manual for the XGE implementation, but more a summary of how various interface paradigms may be achieved through use of the system.

The goal is to provide a full graph editor, using the filter stream mechanisms to provide a flexible user interface. In practice, a more general approach has been taken, to provide a framework by which specific graph editors can be implemented.

The interface provided can be thought of as forming three layers. The bottom, layer provides means by which graphs can be formed out of networks of objects, and how these graphs can then be manipulated as structures.

An intermediate layer provides means by which the graphs can be mapped into visual displays, using filter streams.
The upper layer provides a means by which a user interface can be built on top of this structure. It allows graphs to be manipulated from the visual end of the filter stream, as opposed to the structural end.

11.2 Example Implementations of Common Idioms

This section describes how a number of popular user interface widgets can be implemented using the upper layer of the XGE system. This is by no means a definitive account; the system is powerful enough that each could be implemented in any of a number of ways. The purpose is solely that of demonstration.

In the examples that follow, code fragments make only half-hearted use of the fully general transformational concepts of the system. This suffices to give an overview of the ideas. In each case, a “purer” implementation could use virtual structural transformations to provide everything. There is no advantage to describing these in great detail.

11.2.1 Menus

There are two obvious ways of using menus in the XGE system. The first is simply to use the built-in menus provided explicitly by the system, using the CreateMenu, DestroyMenu and PopupMenu calls.

However, the types of menus that can be produced with this is rather limited. It is provided more for ease of use than for any better reason. A second, much more powerful method is to use the graph-manipulation potential of the system — treating a menu as simply a special form of graph.
A menu can be produced from scratch by designing a special graph form, in which the objects of the graph are the "buttons" in the menu (with appropriate labels on them to display their purpose). The use of general labels means that a button can be labelled by anything — an arbitrary picture or diagram, rather than simply a text string.

There is a problem in this that the Pause function provided by the system knows about internal menus, but would not know about graph-based menus. This could be tackled by putting a menu-detector into the key-bind definitions for the mouse buttons, so that clicking while in a menu graph would cause the appropriate actions. This is the only way provided by the system for implementing context-dependent actions.

Given this, the menu-driving functions may then be written in terms of the creation and destruction of graphs. One great advantage of this is that menus are no longer static entities — they may be subject to change and filter-distortion, just like any other graph. Thus, one menu may appear different depending on the context in which it is called, without the overhead of having to produce new menus on the fly.

A pure implementation could, of course, implement menus by displaying them as "virtual objects" that are interposed by a menu-providing streams module in the window stack.

### 11.2.2 Folds

The concept of a fold, such as is found in folding-editors, is merely a particular form of transformation to present data-hiding. This being the case, it can be implemented simply in terms of filter modules.
Folding is the visual combination of several objects to appear as a single object (whether the "several objects" be characters, lines of text, subdiagrams, or whatever). Once a representation for these is produced in terms of graph descriptions, a filter module can be produced to implement the appropriate kind of folding, using a combine-type module.

Since the system provides no support for persistence, the problems dealing with persistent folds (i.e., folds that remain between separate invocations of the editing or displaying tool) need not be tackled. This involves the dumping of representations of filter modules into disk files, and their subsequent reconstruction.

11.2.3 Highlighting

This section acts as an example of a particular kind of object specification, from the visual viewpoint. It deals with the issue of showing which objects are being 'selected' on the display at any particular time.

The highlighting of objects is most easily done by putting the direct labels associated with that object into reverse-video mode. The intended behaviour of subobjects is less clear; either they must also be reversed, or they must remain unchanged. The details of how to attain this would depend on the 'colour' specifications associated with these objects.

The most obvious use of highlighting is in marking what object is currently being selected during a `mf_position` or `mf_object` call. For this reason, these calls can take `start-highlight` and `end-highlight` functions in optional parameter lists. Of course, these functions could be used for other purposes, such as selection-restriction. To highlight an object when it is available for selection, it suffices simple to pass a selection-validation function (`start-highlight`) based on the following code fragment. It acts by calling the local `hilitelabel` function on each label of a
selected object, which causes the label subobjects field of each to be replaced by a new value, which results in the colourmap entry being reversed.

```plaintext
fun highlight (on:bool) (obj:Id) = 
  let val labels = GetObjectLabels (obj);
    fun hilitelabel (ob:Id) =
      let val labsub = GetObjectSubObjects(ob);
        val (labview,labdata) =
          case labsub of
            el_labelsubobjects stuff =>
              stuff
          | _ =>
            raise panic with
            "not label";
        val {view_id,view_size,view_position,
             view_colourmap} = labview;
        val newlabview = {
          view_id=view_id,
          view_size=view_size,
          view_position=view_position,
          view_colourmap =
            if on
            then COL_REVERSE
            else COL_NORMAL
        };
        val newlab =
          el_labelsubobjects
          (newlabview,labdata);
    in
```
SetObjectLabelSubobjects
  (ob, fn x => (newlab,[]))
end

with a call such as:

    mf_object([highlight true, highlight false]);

A pure implementation could provide automatic highlighting of objects within a particular streams module in the window stack, that is set up to virtually transform the colouring of particular objects, in response to messages from the top-most user interface level.

11.3 General Remarks

11.3.1 Object Oriented Methods

The closest it is easily possible to get to object-orientation is "templating". This is a means of providing sets of template objects, and building up graphs in terms of "clones" of these templates. This has the advantage of allowing assembly of very complicated objects in advance, and then using the cloning process to produce variants.
The cloning is achieved by simply copying the appropriate attribute specifications into a newly created object. A form of “edited” cloning can occur by selective attribute copying.

This attribute-based cloning mechanism is not explicitly provided by the system, but is more a byproduct of the way things are organised. It works particularly well in conjunction with the lazy-evaluation method for creating arbitrarily unlimited graphs, as the cloning may occur at the time when an object is instantiated.

An obvious way of creating such clones is by selection from a menu of template types, followed by attribute-copying when the object is actually created. This is a particularly apt method when constructing an entity out of a collection of components of various types. The component type is selected from a menu, resulting in identification of a template object for a component of that type. An object can then be created with \texttt{NewObject} as a copy of this template. Any alterations to this object will be local — the original template will remain unchanged.

This is in fact a particularly powerful scheme, as these newly created objects may themselves act as templates for other new objects.

There are only two potential difficulties with all this:

- \textit{self-referencing}.

  None of the attributes of a template object may reference any template object (including itself). If any attributes do this, then an instantiation of the template must involve changing such references so they point at suitable cloned objects — not the templates themselves.

  In particular, any attributes such as subobjects, labels, \textit{etc.} must be recursively template-cloned when the object is instantiated, so that the subjects, labels, \textit{etc.} of the new object are themselves new. Failure to check for this will result in a graph with multiple occurrences of a single object, leading to
disaster. (In fact, it is not quite as bad as this — the system will panic due to internal inconsistency, so it will be immediately obvious what went wrong.)

- *template appearances.*
  A template object must not, for the reasons outlined above, itself appear in a graph; only instantiated instances of it may appear.

### 11.3.2 Functional Methods

Functional methods are most useful in the system when used in terms of the specification of object attributes. The system provides only a very rudimentary (but powerful) attribute specification language — namely just the evaluation of functional expressions.

Any specification language (within reason) could be superimposed by providing a mapping between interpretation of terms of this language onto raw functional specifications. No such libraries are provided by the system itself, though these should exist as it is clumsy to use these specifications in raw form.

There follow a few examples of the kind of specifications that might be used to define object attributes:

- *Constant Specifications.*
  The simplest form of attribute specification is a constant function, such as the `Attribute`-typed position expression

    `( { ofield_spec = fn (_,d) => ( ) ) } )`
el_position(pos_value(0,0)),

d
),
ofield_depend = nil,
ofield_value = el_null,
ofield_fuzz = (el_null, el_null)
},
dep_position
)

which is a specification for a constant (0,0) position attribute.

- **Dependent Specifications.**

A non-constant specification must be dependent on some other attributes, the parameters with respect to which it varies. For example, the size of a particular object might depend on the sum of the sizes of its subobjects — the more subobjects there are, the larger the object should appear. This can be specified with an expression such as:

```plaintext
(
{
    ofield_spec =
    fn (i,d) =>
        (el_size(size_value(kidsizes(i))),
        d),
ofield_depend = (obj, dep_subobjects),
ofield_value = el_null,
```


ofield_fuzz = (el_null, el_null)
}

dep_size
)
where kidsizes is some function which evaluates the size of the object by adding up the sizes of the children of that object in some way.

Whenever the subobjects component of the object obj is modified, its size component will be automatically re-evaluated since the dependency (ofield_depend) part of the size specification lists the subobjects of obj.

Note that the object obj must be named explicitly by its descriptor. Since one attribute of an object should quite frequently be dependent on another attribute of that same object, it would probably have been helpful to provide a self descriptor, by which an object may refer to itself without having to know its own name.

- Complicated Specifications.
  Since specifications may contain arbitrary ObjectFieldFunction function components, they may be arbitrarily complicated. It is best to bear in mind that the more complicated the dependencies involved, the longer the system will take to resolve them.

11.3.3 Generating Displays

An object is visible on the display only by virtue of its associated label attachments. Since the labels of an object are simply another attribute of that object, the appearance of an object can be manipulated and specified as with any other attribute.
This is a particularly powerful technique, as it does not impose any artificial restrictions on how objects may appear. Of course, the more complicated the appearance is, the greater the overhead in terms of label attachments.

As examples, there follow a collection of simple-minded object displays.

• **Borders.**

for the outline of an object to be visible, a bordering label must be associated with it. This is simply a label object the same size as the object itself, whose contents are a collection of lines around the edges. The system does not at present provide for these lines being of arbitrary width — they are just narrow single-pixel lines.

Such a border label should have its position and size specifications as:

\[
\text{ofield_spec} = \text{el_position(pos_value(0,0))};
\]
\[
\text{ofield_spec} = \text{el_size(size_value(1000,1000))};
\]

and its picture specification as

\[
\text{ofield_spec} =
\]
\[
\begin{align*}
\text{el_picture( MakeViewer(false),} \\
\text{label_diagram} \\
\text{diag_lines =} \\
\text{[[} \\
\text{(0,0), (0,1000),} \\
\text{(1000,1000), (1000,0),} \\
\text{(0,0) } \\
\text{]]}, \\
\text{diag_curves =} \\
\text{nil}
\end{align*}
\]
This will produce a label with a border diagram around its edge. It might be useful to put this border picture in the `el_picture` component of the object itself, rather than using a separate label object for this purpose. It also might be wise to have the lines slightly within the frame, rather than on the very edge, as otherwise problems might occur with round-off error when calculating which lines are contained within which object frames.

- **Text.**

  A simple diacritical text annotation label can be specified as follows. It describes a small label in the middle of an object containing a bit of text.

  This is a conditional label — the text to be displayed depends on some status information associated with the object `obj`, which can be examined with some `whatisit` function, and is stored in the user-definable `el_userdata` attribute of the object.

  - **Position:**

    ```
    ofield_spec = el_position(pos_value(400,400));
    ```

  - **Size:**

    ```
    ofield_spec = el_size(size_value(200,200));
    ```

  - **Picture:**

    ```
    ofield_spec =
    el_picture( MakeViewer(false),
                label_string
                ( case whatisit(obj) of
    ```
HIPPO =>
    "hippopotamus"
}\ BURGLAR =>
    "burglar"
| _ =>
    "alien"
);

ofield_depend = [ (obj, dep_userdata) ];

- Arcs.

An arc object will normally contain just a single line diagram as its picture attribute — a line from the port at one end of the arc to the port at the other.

Arc objects are treated specially in that they are always considered as lying between a port at the top-left of the arc frame and a port at the bottom-right, irrespective of the actual orientation of the arc. Thus, a line between the ports is easily drawn with a picture specification such as

```lisp
ofield_spec = el_picture( MakeViewer(false),
    label_diagram {
        diag_lines =
            [[
                (0,0), (1000,1000)
            ]],
        diag_curves = nil
    }
);
```
As with a border diagram, it is probably easiest to put the arc line in the \texttt{el.picture} component of the arc object itself, rather than using an extra label object to hold it.

11.3.4 Selection

A very common form of object selection is the selection of a \textit{collection} of objects — where the number of objects to be selected is not fixed in any way. This is most easily done by means of a \textit{button menu}, so that objects are selected with \texttt{mf.position} or \texttt{mf.object} and the selection process ended with a special button-press.

The following code fragment shows how this can be done. I use \texttt{mf.position} here, rather than \texttt{mf.object}, because I need to check the \texttt{POSITION} return component to check that the last thing clicked is the "Go For It!" button. (Actually, since I am only providing a single button, this is not strictly necessary.)

```ml
let

(*
 * function to get a list of objects
 *)

fun makeparams () : Id list =

let (*
 * the button to mark the end-of-selection
 *)

val button = {

    menu_prompt = "Go For It!",
    menu_func = fn x => 1
}
```
function to listen for selections until the button is pressed

fun getobjects () : Id list =
    let val f = fn
        x => (GetObjectType(x)=obj_node);
    val (id,(px,_),_) = mf_position([f])
in
    if id=NULL
    then if px==1
        then (*
            * The "Go For It" button!
            * Return an end-of-list
            *)
        nil
    else raise panic with "bad button"
    else id :: getobjects()
end
in

    mf_buttons (place_root, (0,0), [button]);
    getobjects()
)
end;
11.4 Applications

This section describes, in general terms, two possible applications of the XGE system. They correspond to different particular implementations of a graph editor.

11.4.1 Process Modelling

The main application held in mind during the implementation of the system was the graphical modelling of processes — i.e., as a graphical front-end to the Concurrency Workbench [11], a tool for concurrency analysis based on CCS expressions.

This is relatively simple. A process is represented by a node in a graph, and communication between objects is represented by arcs. This communication takes place through named channels, represented by the system as ports.

The Workbench can also produce "state" information, for which the graphical front-end should produce state-diagrams. These are as for process diagrams, except that a process-group "universe" (itself just a node containing internal processes) has incident arcs representing state-changes.

These state-change arcs must be distinguishable from communication-arcs, and this can be done simply by using different labelling for them.

More difficult is maintaining the distinction between state and communication arcs — certain operations may only occur on certain arcs. This is reasonably easy to do (since the details of such operations are provided by the application — the Workbench — itself), but would have been much easier still had the "locking" mechanism been implemented in a much more general way. One reason why no locking is actually provided by the current system is that the proposed mechanism
is so obviously inadequate. Other mechanisms, more in keeping with the streams idea, may be present in the future.

11.4.2 Harel Formalism

Harel [7] has proposed the use of hypergraphs for representing a wide range of "activities". This formalism could be easily implemented using the system — this kind of application would seem an ideal kind for demonstration.

I will describe only one simple example application of this kind — for details of the Harel model itself, see [7].

First of all, a Harel activation consists of a hypergraph in which the nodes are process states and the arcs transitions between states. I will consider only the simple bi-graph case (eliminating parallelism-expansion and nondeterminism). At any given time, a certain set of nodes (states) are considered active, and this activity progresses through the graph by means of the arcs (transitions). A node may contain subnodes (subprocesses) one of which may be the default action — if a transition leads into a state containing subprocesses, then the new state will be the default one associated with this container-state. State transitions may occur either spontaneously or as a result of a signal — since the mere occurrence of a transition causes a signal, a single transition may cause a cascade of associated ones. The driving force behind the model is the occurrence of externally generated signals (caused by user interaction or other "hidden" machinery).

Such a model could be implemented by listening for signals and executing the appropriate transitions in response to them. The XGE system would do a lot of the work involved by means of attribute dependency specifications. The only attributes involved will be the picture attributes (controlling visual details of the display, such as highlighting of active processes, incoming signal events, etc), and
the **userdata** attributes (to store information regarding the current state of each process). The specifications of these attributes could be expressed statically, so that XGE could perform the entire model simulation on its own, with no support from a "driving application" necessary.

Of course, this would be an unnecessarily complicated way of implementing a particular Harel simulation — and would need to be re-implemented for each such simulation. More useful would be a general interpreter for Harel simulations, acting as a driving application, and loading in attribute specifications on demand.

Note that, in principle at least, it would be hardly more difficult to produce a "static" system such as this (in which the graphs themselves are fixed, and only their status is changing) than to produce a dynamic model, in which new objects (processes) can be created and destroyed as the simulation runs its course — such as would be necessary for a CCS-based simulation.

### 11.4.3 Imposition of "Views"

The most interesting component of the system is the streams-based design of the "views" implementation. It is also the most difficult part to describe in totally general terms, without reference to any particular structure being imposed on the data forms being transferred. Here is one place where the polymorphic typing of **Standard ML** should be useful — the building of general library functions for use in constructing view filters. Total generality cannot of course be achieved, due both to the less-than-adequate implementation of filters, and also to the very nature of the problem; in order to make structural changes, there must be at least some idea of the kinds of structures involved. An object-oriented functional language would have won in these terms.
Under these limitations, this section will describe merely the kinds of things that can be achieved with filters, with very little detail as to how these would actually be implemented. The details of the implementation will depend heavily on the structures involved in the graphs being manipulated.

Here are short examples of how various view-based ideas could be implemented:

- **Hiding.**

  Operations may be applied to a collection of objects simultaneously by hiding the collection behind a single virtual object and applying the operation once to this. Of course, in order for this to be viable, all such operations must be uniform.

  Hiding of this kind is implemented using a combination filter, to combine all of a collection of objects into a single virtual object. Operations on this virtual object are converted into a collection of operations, each acting on one of the components, when the action is reverse-filtered from the target window back to the base graph.

  The following code fragments give an idea as to how this combination might be managed in an application:

```plaintext
(*
   * create a filter
   *)
val hide = NewFilter([ {
    conv_type =
      filt_combine {
        fcomb_distort = nil,
        fcomb_newobj = nil
      },
```
conv_match =
    match_targetlist
    [o1, o2, o3, o4 ],
conv_action =
    (nil, nil)
}
]));
(*
 * push this filter onto the window stream
 *)
AddFilterAssoc(windowfilter, [hide]);

(*
 * do the operations
 *)
do_operations(...);

(*
 * pop the hiding filter off
 *)
RemoveFilterAssoc(windowfilter, [hide]);

(*
 * clean up the dead filter
 *)
DestroyFilter(hide);

Note that this example uses the simplest possible form of combination filter. A real example would need to be a lot more sophisticated — the filter type
conv_type would perform some distortions to the objects being combined, with fcomb_distort and the new object being created would have some non-default attributes set, with fcomb_newobj; the objects being combined would be prompted for, rather than by using a fixed list, with conv_match; the filter could actually do some work, rather than simply passing messages through, unchanged but for the "narrowing/broadcasting" imposed by the combination, with conv_action.

- Folding.

Folding is like combination, with the exception that the objects being combined are no longer visible in the image window. There are many ways in which this may be implemented, but the easiest is probably simply to turn off subobject-instantiation in the virtual combine-object. Unfortunately, the current implementation does not provide for filtering of "additional" attributes of this kind (those not of type ObjectField), so instead a virtual delete of the object will have to suffice. In fact, this will be more efficient, since the folded objects will not exist at all in the virtual graph.

Thus, the filter would be produced with a conv_type of

```haskell
conv_type = filt_combine {
    fcomb_distort =
        [ fn _ => fmod_delete ],
    fcomb_newobj =
        nil
}
```

The number of virtual operations that may be performed using filters is almost limitless, so for brevity I will provide only these two small examples.
12.1 Limitations of the Current Implementation

12.1.1 Design Limitations

Almost all of the XGE system has been designed in a way that is a compromise between power of expression and ease of use. Thus, nothing is quite as general in concept as would otherwise be possible.

A pure concept of "object" would be that an object is totally described by a list of attributes — where these attributes are dynamic quantities, whose interpretation is left entirely to whatever operators are called on them. I have hard-wired special attributes. This forces hard-wired interpretation of the special attributes by the entire system, but has two good points:

- It makes the system a lot easier to use as a programming tool for building interfaces which are modelled on concepts suitably close to the hard-wired attribute design.
On the other hand, more complex interfaces are made a lot more difficult to implement.

- It allows the system to run at a reasonable speed.

A more mundane problem lies in the choice of X-windows as the display-manager. XGE requires a specially modified version of Standard ML, with support for the X Library. X10 was used, as X11 was unfortunately not available at the time.

As an efficiency measure, the system itself does not use X to the full extent that it could — only graphs are implemented in terms of X objects — the individual objects are not. It would have been significantly simpler to implement every system object as an X object, but also horrendously inefficient — X is not particularly adept at handling hundreds of inter-related objects in real time.

### 12.1.2 Implementational Limitations

Given these limitations, several useful features are not provided by the current implementation, though they could easily be incorporated. These are:

- **Locking.**

  The scheme provides hooks for “attribute locking”. This restricts certain attribute-modifying operations to be applicable only in certain circumstances.

  This is useful, in that it allows the use of totally general “editing” facilities, in a consistent manner. When a more limited kind of editing is required (such as applying only to a certain class of objects, for example), then the same general facility can be used, but with only that class of objects made available to it.
Locking is done at the object-level, rather than being a feature of the interface between acting operations and object operands.

The following possible locking mechanisms would provide a more general framework.

- **operation-based locking.**
  This would cause a lock to be a feature of an operation rather than of an object. It would of course require a system to maintain levels of privilege with respect to various operations, if done properly.

- **view-based locking.**
  More in fitting with a streams-based system would be a streams-based locking mechanism. This would involve the use of “protection filters” which could be pushed onto graphs in the same way as “view modification filters” are at present. This seems natural — this kind of locking/protection is a relative rather than an absolute thing, in that an object locked in one respect may still be modifiable in another.

- **Object Windowing.**
  Windowing views, for use in virtual operations such as zooming, panning, etc, are provided only in rather a patchy form, and do not stand up to much misuse.

  Ideally, details like this should be handled by the windowing system itself (the X system, in this case). However, for a mixture of reasons, this is not the case. But it nonetheless seems wasteful to have to implement such things internally within the XGE system itself, so I provide only a very rudimentary capability.

- **User-Defined Attributes.**
  I provide only simple string-to-string associations to allow additional user-
interpretable attributes to be associated with objects. A much more pleasant scheme would be to allow typed attributes.

This would probably be implementable in a rather object-oriented way, in that a user-defined attribute would consist of a group of functions to be associated with the object, each performing some task in the interpretation and use of the attribute.

Unfortunately, to be generally useful this would require the association of a list of groups of polymorphic functions, where each function-group in the list would deal with a different type. This could of course be simulated using datatype types, but would be less than satisfactory.

For these reasons, I have left this region well alone and have provided very little in the way of support for such things. In the worst case, the current implementation may be used to simulate more complex schemes, in that various types may be encoded into strings before being associated with an object, and decoded when required.

12.2 Redesign Issues

One conclusion to be drawn from this work is that not much is gained by the use of a pure functional programming language for the specification of a streams model. Certainly, it provides an extremely convenient notation for expressing the transformations involved, but this is achieved at the expense of identifiable structure.

The problem arises from the need to pass functional values through a streams filter. Since such a filter operates by making alterations to the data being passed through it, based on the structure of this data, it is a great drawback to be unable to examine this structure. Functional values are opaque, thus making this impossible
in a pure sense. The current implementation cheats by associating extra information with the data, which the filtration can examine. This has the disadvantage of freezing the types of structures which can be passed through the filters — leading to an inextensible system.

This is unfortunate, as use of a functional language is particularly apt for the implementation of streaming. The problem comes from the need for a simultaneous "functional look" to both the streams themselves and also the data being subject to streaming.

This data arises from the very generalized model of graphs. Unfortunately, a certain degree of generality has had to be sacrificed in order to permit streaming. On reflection, a superior method would have been to implement the functional components of graph attributes in terms of symbolic interpretation rather than of functional evaluation. Being symbolic, values of this kind could quite safely be passed through fully general stream filters.

A natural next step would be to implement the streams mechanism itself in purely symbolic form. This would allow the use of meta-streams of arbitrary levels, all implemented symbolically.

Even at this stage, many benefits of a functional language still remain. Thus, it is by no means clear that implementation using an object-oriented language would be much of an improvement, though object-orientation is a natural way of thinking of both general graphs and also of general streaming.
12.3 Optimization

The major downfall of the current implementation is the lack of optimisation available in the following areas:

- **the display.**
  Redisplaying of graphs due to internal changes occurs at a very gross level — almost any kind of alteration causes a rippling effect, resulting in almost the entire window being cleared and redrawn. An intelligent optimization strategy would greatly speed up this process.

- **filtering.**
  As no temporary "mid-stream" objects are kept between one occurrence and another of a stream filtering, a significant amount of additional streaming occurs needlessly.

  Were the mapping of a stream between a window and a graph to include the creation of a shadow stream-tree, mirroring the structure of the stream but containing temporary object states, then only those objects which had truly changed would need to be refiltered.

  This would also fix a problem in the current implementation, where `combine` filters are involved. At present, it is necessary to re-filter from scratch all objects which may be subjected to a combination filter at some stage, since the combine operations require the existence of the complete set of (anonymous) objects being combined at that stage. If such temporary objects were already in existence, then this would not be a problem. At present, this issue is not addressed at all, meaning that problems may arise if an object is modified and is then combine-filtered by a stream at some level. This only matters
for "non-constant" forms of combination, where the resultant combination objects depend on all of their constituents.
Chapter 13

Conclusions

13.1 Summary

13.1.1 Graph Representations

A general form of graph representation is presented, similar in external structure to the Harel hypergraph form [7]. Its internal structure resembles a process model — a graph is represented as a network of objects, each with built-in specifications of responses to events (ie, "programming").

Each object in a graph is provided with a descriptor (or "handle") by which other objects may refer to it, in order that the network may be specified. Every object is defined in terms of a set of attributes, each of which is separately specified by a Standard ML expression.

A system built up from a graph network of objects is dependency-driven; a change of state of an object may trigger a change in state of other, dependent objects. This leads to a fully dynamic graph model.
The XGE system implements a subset of the full functionality that could realistically be achieved by such a model. Attributes are hard-wired, rather than specifiable in terms of an attribute type specification language, for simplicity. Not all attributes are fully specifiable in terms of the general attribute specification mechanism — again, this is for simplicity and efficiency.

### 13.1.2 Filter Streams

Graph networks are accessible in terms of streams filters. These are series of transformation channels to be applied to the objects of a graph to produce an altered image of the graph. A filter consists of a string of modules, each of which performs some kind of transformation.

The end result of filtering a graph through a stream is a new graph, based on the structure of the original graph, but with any amount of possible alterations brought about by module action. A module can be considered as a function acting on object-specification packets to produce new packets.

The new, filtered, graph network is built up from new descriptors, so that the structure of the new graph can be derived entirely from the structure of the old graph and the structure of the filter modules.

### 13.1.3 Graphical Display

A display of a graph is produced by mapping an internal graph structure into an external graph structure set up in a window, by means of a filter stream. Alterations to the base graph result in the passing upstream of new object specifications, to update the window graph; alterations to the window graph result in object specification being passed downstream through the filter to the base.
Many windows may be attached to the same base graph with different filters. This permits multiple different views of the single graph to be available (such as simultaneous viewing of different aspects of the graph).

### 13.1.4 User Interface Specifications

A user interface may be specified by construction of menu and key-stroke bindings. Such a binding causes evaluation of some **Standard ML** expression whenever the appropriate menu item is selected or the appropriate key-stroke occurs.

This expression evaluation results in some action being taken by the system. This can occur either by means of direct graph manipulation (alteration of a windowed graph), or indirectly by means of communication with an underlying application program, leading to eventual alteration of a base graph (and thus to filtered alteration within a windowed graph).

### 13.1.5 Construction of Interactive Graphical Systems

A graphical system may be constructed by attaching a user-interface specification (**ie**, a menu/keystroke binding table) to an applications program (**ie**, the "driver" to which a graphical interface is required).

This driver application must produce internal graphs which are then mapped onto windows for display and control. Interaction may then proceed as specified by the bindings table.

As an example of this paradigm, the following steps are carried out when the **XGE** graphical system is used. Details of the implementation of **XGE** are given in the appendices following this chapter.
• First, a central layer performs any initialization that may be required (such as setting up of initial graphs, windows, filter streams, interaction bindings, etc).

• Then the central layer calls Interaction, which causes an interactive session. During this, menu/keystroke bindings table entries are executed in response to user interaction.

These entries either call direct XGE functions to effect the display system, or else call driver-level functions to effect the underlying application. The driver can then call XGE functions to manipulate internal graphs in response to this.

• When Interaction returns, the session is completed and any cleaning-up that might be needed must be done before exiting.

13.2 Remarks

13.2.1 Advantages to the Approach

There are two main concepts that characterize the approach taken. These are: the representation of a graph in terms of a self-contained network of attribute specifications, linked together by a descriptor-based object-identification scheme; and the streams method of structure transformation to present a display interface, imposed on this graph representation.

The most obvious advantage is the presence of structure and form at all levels of the system. This automatically leads to a much more modular and extensible architecture.

In principle, this structuring of the system leads to the possibility of optimizations; a conventional unstructured system cannot be optimized (except in terms
Chapter 13. Conclusions

of optimizations which are themselves unstructured, and therefore not suitably
general for all cases). Optimization is made even more plausible by the use of a
dependency-based graph attribute representation; local changes must remain local,
and therefore optimizable.

13.2.2 Disadvantages to the Approach

Unfortunately, this very structuring also leads, unless much effort is expended, to
lack of efficiency; it is much easier to optimize for one specific unstructured case
than it is for a general form of structured cases.

13.2.3 Implementation and Language Issues

At first glance, it might appear that a functional language would be ideally suited
for implementation of a streams system. Unfortunately, a trade-off must be met
between generality of expression of graph representations and generality of expres-
sion of stream transformations. Any representation that involves constructs other
than strictly identifiable terms raises problems for a fully general streams-based
transformation system. This is hardly surprising, as such transformations are anal-
ogous to pattern-matching; this cannot work for terms from which no structural
"pattern" can be extracted.
Bibliography


133


