A Method for Establishing Informational Equivalence among Forms-based Interfaces

by

Simon R.Rollinson

Submitted in accordance with the requirements for the degree of Doctor of Philosophy.

The University of Leeds
School of Computing

March 2003

Abstract

This thesis presents a graph-oriented approach to modelling, and comparing the equivalence of, forms-based data entry and retrieval interfaces. We focus, in particular, on the type of forms used in graphical user interface environments and show that several types of equivalance exist amongst them.

The forms interface schema (FIS) is developed for the purpose of modelling forms. The graph-oriented nature of the FIS enables differences in the presentation, caused by different windowing environments, to be ignored whilst maintaining the presentation invariant properties, such as the relationships between interface components.

A method for extracting the data model underlying a forms interface is presented along with a collection of transformations based on a graph rewriting paradigm. The transformations establish a mapping between user interface components and the primitives of an extended Entity-Relationship (EER) model. It is shown how the mapping of interface components is contingent upon their use and we describe a mechanism for accommodating this.

Approaches for detecting equivalence among forms interfaces are investigated. We show that interface components can be placed into classes and how, under certain conditions, it is possible to use components from the same class interchangeably. More complex types of equivalence are supported by the mapping of forms interfaces to EER schemata. It is shown that two form interfaces are equivalent if they map to the same EER schema, or if there exists a canonical EER schema into which the schemata of a pair of interfaces can be transformed via restructuring rules.

Acknowledgements

First, I would like to thank my supervisor, Stuart Roberts, for his advice and encouragement and most importantly for keeping me thinking when I thought I had run out of ideas. I would also like to thank my advisor, Peter Mott, for the initial discussions which gave me the confidence to undertake this research.

I am indebted to my wife, Sarah, who has provided a constant source of love and support that has sustained this research. Thanks also go to mum and Ian for their support.

Friends have played an important part in this research in one way or another. Thanks, therefore, go to Chris Hemingway, who I shared an office with for many years and who provided an endless source of encouragement. Thanks also to Andy Fletcher who was always able to inject some humour into the proceedings.

Finally, a special thanks to my colleague and close friend Mark Steer who, over the past several years, has provided an invaluable source of support, guidance and advice.

Dedicated to the memory

of my father, Richard Rollinson.

Declarations

Some parts of the work presented in this thesis have been published in the following articles:

Rollinson, S. R. and Roberts, S. A. Formalizing the informational content of database user interfaces. In *Conceptual Modeling - ER'98, Proceedings of the the 17th International Conference on Conceptual Modelling, Singapore, November 1998*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The importance of user interfaces to database management systems was recognised by the database community in the early 90's (Silberschatz, Stonebraker, and Ullman 1990) and despite a report three years later critizing the lack of progress (Stonebraker, Agrawal, Dayal, Neuhold, and Reuter 1993) a significant amount of research has been undertaken in this area. This is evidenced by the "User Interfaces to Databases" workshop series (Cooper 1992; Sawyer 1994; Kennedy and Barclay 1996) and more recently in the User Interfaces to Data Intensive Systems workshop (Paton and Griffiths 1999). Database interface research has resulted in a variety of methods for data access, data definition and also tools for building database user interfaces. Not suprisingly, a considerable amount of effort has been directed toward the problem of simplifying access to data stored in database management systems (DBMSs) for novice and casual users. The most well known of these approaches was Query-By-Example (Zloof 1975) which, despite its age, still proves popular in today's desktop database management systems. The development of semantic data models generated interest in providing database interfaces that employed a *browsing* paradigm rather than the traditional query access. The Entity-Relationship model (Chen 1976) was the focus of several efforts in particular. ARIEL (Burns, Malhotra, Sockut, and Whang 1993), for example, allows the user to navigate an Entity-Relationship (ER) diagram by following relationships. Approaches to developing data definition interfaces have also employed semantic data models (see Chuang and Springsteel (1989) for example). More recently virtual reality techniques have been employed in database interfaces to allow the visualisation of complex data sets such as those found in environmental monitoring (see Wittenbrink, Rosen, Pang, Lodha, and Mantey (1995) for example).

A separate strand of database interface research, distinct from data access and definition interfaces, has considered the problem of creating database interfaces. In particular, this work has focussed on the automatic generation of interfaces using declarative models of the data to be represented in the interface and the tasks that are to be performed on the data. Model-based systems, as they are called often use a semantic data model to describe the data from the database that is to be used in the interface. Early approaches such as MACIDA (Petoud and Pigneur 1990), TRIDENT (Bodart *et al.* 1990) and GENIUS (Janssen, Weisbecker, and Ziegler 1993) used the ER model and were able to generate a collection of forms to support data entry and update to the database described by the ER model. More recent model-based approaches have extended these systems and are capable of generating several types of interface (see for example Griffiths *et al.* (1998)). Despite the success of model-based systems in the laboratory environment, they have not been widely received in industry where the most common tool for interface construction is the GUI builder (Griffiths *et al.* 1999). Furthermore, many of the commercial interface generators limit themselves to the construction of form interfaces because of their popularity in business environments.

Whilst the fundamental aim of research into database interfaces for data access and definition has been to simplify these tasks for novice users there has been little attempt to address the problem of allowing end-users to construct their own data access interfaces, or at least modify an interface that has been generated by a GUI builder, for example. The majority of approaches to constructing database interfaces require the user to possess a significant amount of experience in manipulating semantic data models. Clearly such skills cannot be expected of novice users. Indeed, it has been shown that semantic data models such as the Entity-Relationship model are difficult to understand by end-users (Moody 1996) and there is also some evidence to suggest that database practitioners find these models difficult to work with (Hitchman 1995). It is suprising to find, therefore, that even approaches to constructing database interfaces which are aimed at novices, such as that described by Pizano, Shirota, and Iizawa (1993), still require the user to manipulate a semantic data model in order to construct the interface. A more promising approach is reported by King and Novak (1987) who propose the Freeform system that enables end-users to modify forms-based data entry/update interfaces. Freeform is based on a *drag-and-drop* paradigm in which the user interacts directly with the fields on the form. By manipulating the form in this way users do not have to deal with an unfamiliar semantic model and can visualise the effects of their modifications immediately. The modifications are also validated to ensure that a user's personal copy of a form maintains consistency with the original master copy of the form. The consistency between forms is enforced by constraints specified on a master form by the form's designer, who is usually an expert. Despite its benefits, Freeform suffers from the limitation that it is only able to operate on individual forms. Modifications such as decomposing a form into several forms or merging several forms into a single form are, therefore, not permitted. The implications of this are that any modifications made to an interface must preserve a one-to-one correspondence between forms in the original interface and forms in the modified interface.

The lack of research into supporting novices constructing and altering database user interfaces is suprising given the the importance placed on such facilities by both the database and human computer interaction communities. Myers (1995), for example, argues that end-user customisation facilities are an essential part of the user interface, whilst England and Cooper (1992) recognise the need for end-user customisation in the form of *reconfiguarable* user interfaces for databases. Despite the encouraging results of some of the approaches discussed above, little has been done to address their deficiencies. The work presented in this dissertation addresses these deficiencies, and does this within the wider context of improving information systems development tools.

To illustrate how the work presented here can be used to support information systems development, consider for example, revision control systems such as the Concurrent Versions System (CVS) (Cederqvist *et al.* 1992) and Microsoft Visual Sourcesafe (Microsoft 2001a). These tools play an essential part in the development and maintenance of software. Revision control systems allow developers to track changes and compare and manage versions of source files. A problem with current revision control systems is their bias towards programming languages such as C and C++ whose source files comprise simple ASCII text for which many comparison techniques exist. Consequently, such tools lack support for "Visual" languages such as Visual Basic (VB) (Microsoft 2001b) and Delphi (Borland 2001) that rely on a direct manipulation paradigm for constructuring applications based around form interfaces. As a result of this, users of these visual development tools cannot take advantage of the facilities provided by revision control systems, such as “diff” tools, that allow different versions of source files to be compared, for example. It is possible to envisage the situation in which a Visual Basic developer, working in a team environment, would want to compare two versions of a VB project that had been worked on by several developers. Any comparison must treat the interface as a whole rather than on a form-by-form basis. Without the ability to do this, changes such as merging two forms together could not be detected.

A major attraction of VB to many people, especially those with little programming experience, is the speed and relative simplicity at which form interfaces can be built when compared to traditional programming languages. A common component of many VB applications is a database that is used to store the information captured through the forms. For many end-users, constructing a database is a difficult task, requiring expert knowledge when compared to the simplicity of building forms. A more natural way of working would be to hide the database from the user and build this automatically based on the forms they create. Such a facility would, in effect, work in the opposite way to the “form wizard” in the Microsoft Access DBMS. Thus, rather than generating a form based on a database table, database tables would be generated based on forms. To provide this level of functionality, it is necessary to understand how to translate information structures at the interface level into structures that can be represented in a DBMS.

A further application of the ability to recognise information structures present in a form interface is for reverse engineering. Consider the scenario in which a legacy application is to be replaced. Before it can be replaced, it is necessary to understand the information captured by the application. As with many legacy systems little documentation exists and the format of the files in which data is stored is unknown.

However, the application employs a form interface for data capture. If reverse engineering tools provided the ability to analyse the form interface, to extract the information structures, then this would provide the appreciation of the information captured by the application that was required.

There are two key requirements of all the applications described above: an understanding of how information is represented in form interfaces; and a formalism for describing this information. In addition, some means of being able to compare this information, to identify situations in which two form interfaces are different and when they are equivalent is also required. This would provide the basis for a visual "diff" function. The work presented in this dissertation addresses all of these requirements. Using the Entity-Relationship model (Chen 1976) we develop a formalism for representing the information content of form interfaces. We capture the knowledge required to extract the information content in a set of production rules and develop a semi-automatic pattern-matching approach to perform the extraction. To provide an appreciation of the differences in the information content of form interfaces we present a concept of equivalence that is capable of demonstrating the differences, as well as equivalence between form interfaces.

1.2 Background

1.2.1 Information conveyed by form interfaces

Developing a notion of equivalence among form interfaces requires the identification of what should remain constant throughout the modification. Intuitively, this is the information conveyed by the interface. If after modification the information conveyed to the user is different from that conveyed in the original interface then the modifications have compromised the interface in some way. In a form interface the information is conveyed to the user through the fields on the forms. The structure of, and relationships between, both forms and form fields must, therefore, be captured in order to describe the information conveyed by a form. An obvious choice of representation would be a semantic data model, indeed both the IFO model (Abiteboul and Hull 1987) and the Entity-Relationship model (Chen 1976) have been used previously to capture the information conveyed by forms (see Abiteboul and Hull (1988) and Choobineh, Mannino, and Tseng (1992)) and the network model has been used to capture the information appearing in business reports (Holsapple, Shen, and Whinston 1982). It is likely that any model of the information conveyed by a form interface is likely to be a subset of the information conveyed by the database schema that underlies the form interface. The model may also contain information not stored in the database in the form of derived fields such as totals. England and Cooper (1992) refer to the subset of information conveyed by the user interface the *user data model* and recognise, how, during the construction of a database user interface, this model is built through a process of mapping from the internal model of the database. A consequence of this mapping process is the restructuring of the models, often resulting in the user data model becoming un-normalised in order to support the user interface.

1.2.2 A brief history of forms

Form interfaces have been chosen as the focus of this research for a number of reasons. Forms, in both paper and electronic versions, are well-understood and widely used in modern society (Embley 1989). Furthermore, they have become a familiar, taken-for-granted user interface to business operations (Tin-Jun, Machlin, Wang, and Chang 1990). Forms have remained a popular user interface to database management systems for over a decade, despite the many other types of interface that have been proposed. Examples of forms interface range from early character-based interfaces such as Office-By-Example (Zloof 1982) to modern graphical versions of forms such as those found in desktop database management systems such as Microsoft Access and the Oracle Forms environment.

The evolution of forms from character-based to graphical versions spawned a considerable amount of research in re-engineering forms-based applications in an attempt to offset the investments made in character-based systems and the costs of rewriting the applications from scratch (Harrison and Lim 1998). Many of the efforts at reverse engineering user interfaces have focussed on a particular fourth generation language (4GL) or API (application programming interface) due to the proprietary nature of many of the early character-based applications. The AUIDL (Abstract User Interface Description Language) has been developed to enable the re-engineering of character-based COBOL applications (Merlo *et al.* 1995). Using AUIDL, the structure and behaviour of a character-based forms application can be captured through the analysis of COBOL source code. The resulting specification can then be transformed into a GUI specification and at the same time improvements to the interface can be made to exploit the increased functionality available in GUIs. An approach to re-engineering INGRES application-by-form (ABF) applications is reported by Harrison, Berglas, and Peake (1997). Their ITOC reverse engineering workbench analyses ABF source code and the database schema to recover an Oracle CASE specification of the forms application that can be used to generate an Oracle Forms implementation of the application. The future of forms as an interface would also appear secure as the now, well recognised, web-based forms are being extended by the XML community with the proposal of the Extensible Forms Description Language (XFDL) (Blair and Boyer 1999). The aim of XFDL is to standardise a set of form elements and operations which can be used as the the basis for constructing complex forms that can be transmitted securely over the World-wide Web.

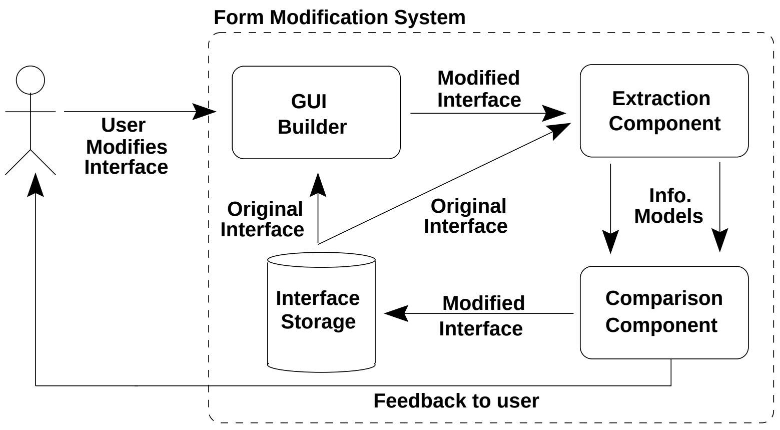


Figure 1.1: forms modification system

1.3 A forms modification system

To provide some context to the work described in this thesis we show how it might be used in a hypothetical forms management system similar to Freeform (King and Novak 1987). Figure 1.1 shows the architecture of such a system and focuses in particular on the modification component of the system that is responsible for dealing with the creation and editing of forms. There are four main parts to the modification system: a GUI builder that allows the user to modify forms; an information extraction component which extracts the information conveyed by the interface; a comparison component that compares the outputs of the information extraction component; and a storage component that holds copies of the valid interfaces for users to modify.

The GUI builder would support a similar functionality to that found in Microsoft's Visual Basic in that it would allow users to *drag-and-drop* interface components onto forms and move interface components between forms. More intelligent functionality would also be provided enabling the linking of forms by dragging connecting arcs between buttons and forms, for example. As with Visual Basic, the addition of a field to a form would cause a label to be added automatically and groups of radio buttons and checkboxes would always be identified as groups by a surrounding groupbox. In addition to these features the GUI builder component would also keep track of identical objects between the original and the modified interfaces thus reducing the amount of duplicated effort required in having to re-establish these correspondences during the comparison stage. Similarly, references to multiple instances of the same object in the same interface will also be maintained by the GUI builder to prevent duplicate effort during the information extraction stage.

The extraction component would be responsible for analysing the original and modified interfaces and extracting the information they conveyed, describing it using some conceptual representation such as a semantic data model. Our aim is for this component to be largely automated requiring little or no user involvement. Any input that is required from the user would be in the form of confirming or rejecting suggestions made by the extraction component. Finally, the comparison component compares the information conveyed by each interface.

In addition to the forms modification system not providing *fill-in* functionality, we also restrict the type of modifications that can be made to the interface. More specifically, the users would only be allowed to make changes that did not require any modifications to the underlying database schema. Thus users would be unable to add new fields to forms that required the addition of fields to the underlying database. We feel that such restrictions are justified given that the target audience of the system are novice users. Whilst this restriction contrasts with Freeform, which does allow users to modify the underlying schema, it is placed in context by the observation made by King and Novak that novice users should be discouraged from making changes due to their lack of experience.

It should be noted that the system described above has not been implemented. Rather, it is described to illustrate the vision of how the work described in this dissertation could be used to simplify the design and maintenance of forms for end users.

1.4 Aims and Objectives

The main aim of the work presented in this dissertation is to investigate methods of capturing and comparing the information conveyed by graphical form interfaces. A particular application, the forms modification system described in the previous section, has been chosen for the purpose of the investigation with the aim being to provide the necessary theory behind the extraction and comparison components shown in Figure 1.1. The focus of the investigation, therefore, will be on how to represent and compare automatically generated interfaces with those modified by a novice user. To this end, three objectives are defined:

- to examine the components and structure of graphical form interfaces to reveal how information is represented;
- to develop a mapping between between an abstract model of an interface and a semantic model; and
- to investigate how two graphical form interfaces might be compared for equivalence

Each of these objectives corresponds broadly to a chapter in the dissertation. Chapters 4 and 5 deal with the problem of representing a graphical forms interface and the information it conveys. In Chapter 6 a graph-based approach to mapping between forms interface and information model is presented, whilst Chapter 7 examines the problem of equivalence.

1.5 Method

In order to achieve the objectives set out in the previous section three tasks were identified. The first of these involved examining graphical form interfaces to identify the types of interface controls used, and the different ways in which controls can be used. The main result of this step is a means of representing graphical form interfaces and their properties.

The second step concentrated on examining how combinations of interface controls are used to represent information and how the combinations of controls map onto structures in an underlying data model. The central aim of this step was to characterise each interface component's ability to represent a particular information structure as a series of mappings from the forms model to a semantic data model.

The final step involved identifying and classifying the conflicts that may occur among graphical form interfaces. Using this classification a framework was devised that was able to determine when two interfaces were equivalent.

1.6 Thesis Organisation

Having identified the requirements for representing and comparing graphical form interfaces the remainder of this dissertation is structured into several "self-contained" chapters. This somewhat unorthodox structure has been adopted due to the approach taken and the scope of the material required to solve the problem. Each chapter, where appropriate, has its own literature review that reflects the work dealt with in the particular chapter.

In Chapter 3 an analysis of other approaches to mapping from forms to a semantic data model are presented.

An introduction to form interfaces is given in Chapter 2 which looks at how form interfaces are constructed in graphical user interface environments using examples from HTML, Java and Visual Basic forms. The chapter also examines the differences in look-and-feel among different implementation languages and GUIs and looks at how an abstract representations can overcome these differences. The Chapter concludes with a discussion of how the abstraction representations form the basis of the work presented in this dissertation.

Chapter 4 considers the various approaches to modelling user interfaces, paying particular attention to methods that are independent of a specific windowing environment. This is followed by a description of the *form interface model*, a graph-based model capable of representing graphical form interfaces. We also present a *template* form which serves to define the class of forms that can be dealt with by the work described in the dissertation.

Chapter 5 examines semantic data models as candidates for the task of representing the information conveyed by a graphical form interface. Particular attention is paid to those semantic models that have been used previously for the purpose of representing information on forms, and it is shown how hierarchic models are insufficient for the purpose of representing information conveyed by graphical form interfaces.

Having identified conceptual representations of both forms and the information they convey, Chapter 6 presents a graph rewriting approach to mapping from the form interface model to the chosen semantic data model. To introduce the subject of graph rewriting and to illustrate the motivations behind the ideas presented in the this chapter, a review of graph rewriting precedes our approach.

The problem of equivalence is dealt with in Chapter 7 which first identifies and classifies the type of conflicts that may occur among two form interfaces which potentially convey the same information. Having identified these, we focus on the the subset of these which are likely to occur in a forms modification system. A framework for equivalence is then presented which introduces the notion of an isomorphism between form interfaces and the concept of a "normal form" for Entity-Relationship schemata.

Chapter 8 concludes the thesis reflecting upon the approach adopted and identifying areas that would benefit from further study. The potential contributions of the work are also examined.

Appendix A lists the production rules that are used by the graph rewriting system to extract the information content from form interfaces.

A case study illustrating how the ideas presented in this dissertation can be used is described in Appendix B. The primary focus of the case study is the comparison of two form interfaces to an information system, one developed using Microsoft Access, and the other using a scripting language to provide web forms. In addition to this, the case study also considers the use of the work as a reverse engineering method and compares the performance of the method when applied to two different types of interface.

Chapter 2

Form Interfaces

2.1 Introduction

This chapter introduces forms as a user interface by examining modern computer implementations of forms. We focus in particular on several GUI implementations of forms. These range from simple Hypertext Markup Language (HTML) forms to more complex form interfaces developed using Microsoft's Visual Basic and Sun's Java Abstract Window Toolkit and its successor the Swing set of interface components. Having examined different examples of form interfaces the chapter goes on to extract the set of components that can be used to describe a form at an abstract level. This will be done using the concepts from abstract and concrete interface models. With the ability to describe form interfaces in an abstract manner the chapter concludes by describing how such an abstract description can be used as a basis for the work described in the rest of this dissertation.

2.2 Form Interfaces

A form interface to a computer application can be thought of as a collection of data entry forms used to capture data required by an application. The form interface may itself be part of a larger application, such as that used in the case study described in Appendix B, which includes reporting and other functions.

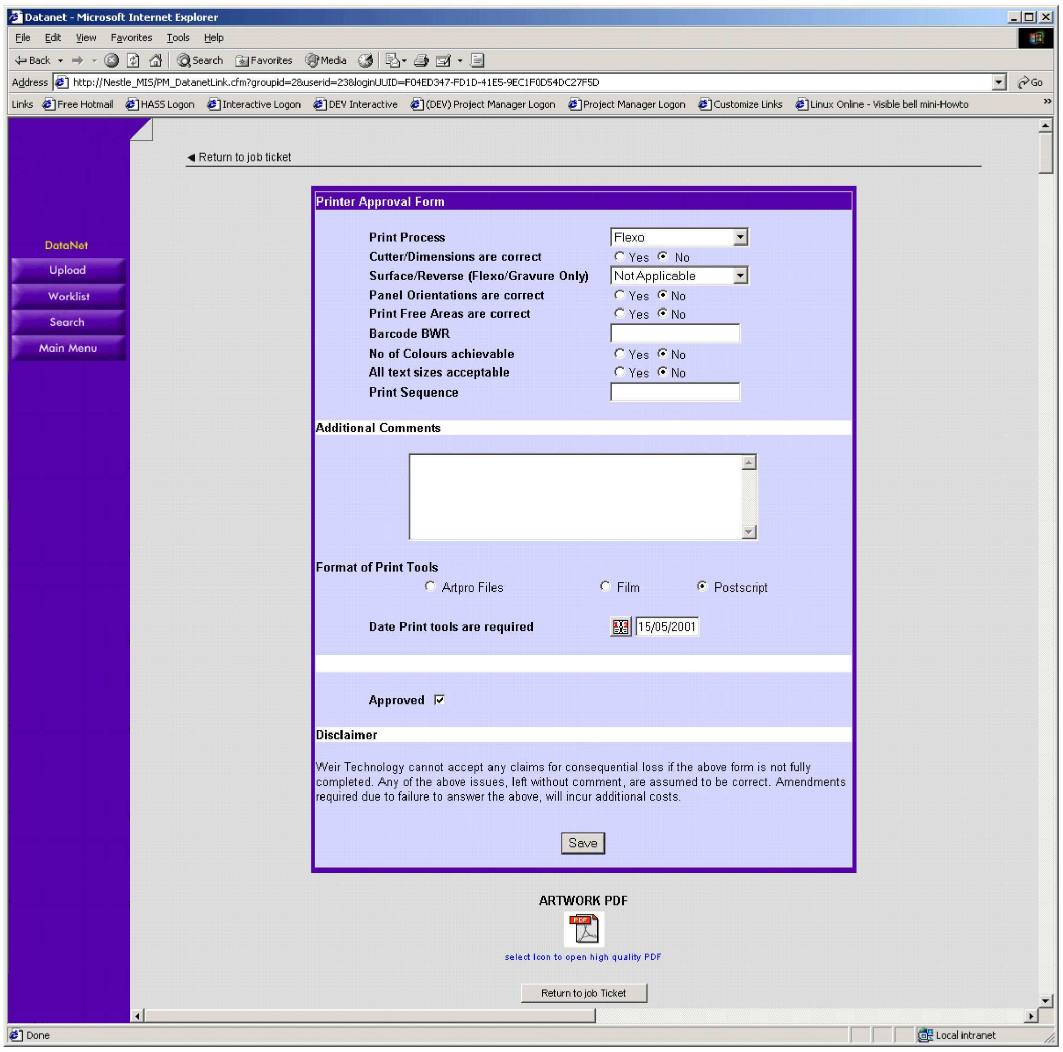
In GUI environments, forms are built from two main components, windows and interface controls (also called interface components). Typically, a window will possess a title bar, containing a description of the purpose of the window and a *main area* which is said to be the *contents* of the window. Windows might also have a menu bar containing drop down menus.

Interface controls can be static or interactive. Static controls, such as a textual labels, or frames, that are used to group other controls, do not respond to any user interaction. Interactive controls, such as buttons or combo boxes, however, are capable of interacting with the user either via the keyboard or direct manipulation.

2.2.1 Forms

A form comprises four parts: an *identifying* part that is used to describe the function of the form; a part containing questions or labels that indicate the information that is required from the form-filler; a part containing the fields in which the form filler will enter their responses to the questions; and possibly an instructional part that provides guidelines as to how the form should be completed. In a form interface, forms are represented using windows, the title bar of which often acts as the identifying part of the form. The questions, fields and instructional part of the form, are represented in the contents of the window. Label controls are typically used to implement the questions on the forms and controls such as textboxes and combo boxes represent the form fields. Other controls such as groupboxes or frames are also used to partition form fields into related groups. For example, one common use of the groupbox is to group together several related radio buttons which offer a mutually exclusive choice. Figure 2.1 shows an example of an HTML form and illustrates the complete set of controls that can be used to build HTML forms. HTML forms are implemented as *pages* and because they rely on a web browser for their output their identifying part often appears on the page rather than in the title bar of the browser window. It is also common for HTML forms to be *long* in that they span several screens. They are built in this way to avoid lengthly delays in processing forms caused by the speed of connections to the web server. Contrast this with forms implemented in Java and Visual Basic which usually restrict the form size to that of the window containing the form.

The interface components available to represent fields in form interfaces depend on two factors: the development language in which the form interface is implemented; and the GUI environment in which the form interface executes. We have already shown the set of interface controls as defined by the HTML standard. In practice, however, these are often augmented by combining the basic controls with other HTML constructs to create more complex controls. Figure 2.2 demonstrates an example of how a grid can be created from an HTML table and several textbox controls. Building such hybrid controls is a complex process and requires a significant amount of effort when compared to languages such as Java and Visual Basic which provide implementations that require no special development. Despite the set of interface controls available for use in HTML forms being limited, HTML forms have the advantage of being cross platform. The only requirement for using HTML forms being a web browser. In contrast, Visual Basic can be used to build complex forms such as the one shown in Figure 2.3 with relatively little effort compared to HTML


Datanet - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://Nestle_MIS/PM_DatanetLink.cfm?groupid=2&userid=23&loginUUID=F04ED347-FD1D-41E5-9EC1F0D54DC27F5D Go
Links Free Hotmail HASS Logon Interactive Logon DEV Interactive (DEV) Project Manager Logon Project Manager Logon Customize Links Linux Online - Visible bell mini-Howto
DataNet
Upload
Worklist
Search
Main Menu
Return to job ticket
Done
Local intranet

Printer Approval Form

Print Process	Flexo
Cutter/Dimensions are correct	Yes [✓ No]
Surface/Reverse (Flexo/Gravure Only)	Not Applicable
Panel Orientations are correct	Yes [✓ No]
Print Free Areas are correct	Yes [✓ No]
Barcode BWR	
No of Colours achievable	Yes [✓ No]
All text sizes acceptable	Yes [✓ No]
Print Sequence	

Additional Comments

Format of Print Tools

Artpro Files Film [✓ Postscript]

Date Print tools are required 15/05/2001

Approved [✓]

Disclaimer

Weir Technology cannot accept any claims for consequential loss if the above form is not fully completed. Any of the above issues, left without comment, are assumed to be correct. Amendments required due to failure to answer the above, will incur additional costs.

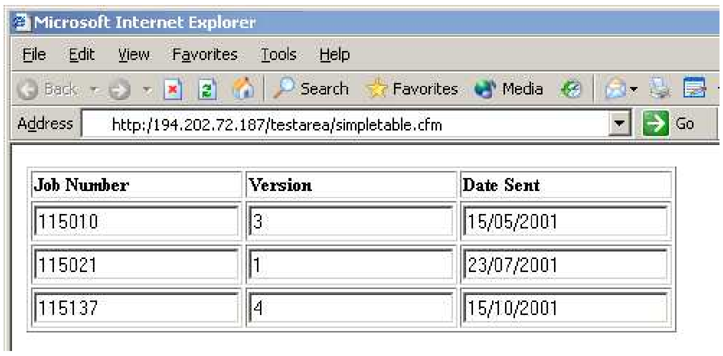
Save

ARTWORK PDF

select Icon to open high quality PDF

Return to job Ticket

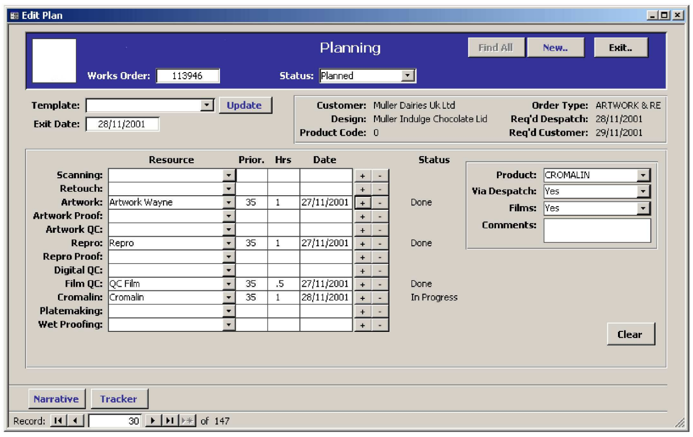
Figure 2.1: A typical HTML form


Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://194.202.72.187/testarea/simpletable.cfm
Go

Job Number	Version	Date Sent
115010	3	15/05/2001
115021	1	23/07/2001
115137	4	15/10/2001

Figure 2.2: An example of an editable table implemented in HTML

forms. For Visual Basic, however, this ability comes at the cost of the forms being less accessible, as they are only available on the Microsoft Windows platform. In terms of accessibility, and the number of interface controls available for building form interfaces, the Java platform is the most flexible. Both the Abstract Window Toolkit (AWT) and the Swing set of interface components are available on all the platforms that have Java Virtual Machines, making form interfaces built using Java accessible to a wide audience. Furthermore, AWT and in particular Swing provide a rich collection of interface controls. It is, however, the Swing components that provide the best cross platform implementation of form interfaces due to the way in which the controls are implemented. Swing components, unlike AWT and HTML controls, do not rely on the host GUI to provide their look-and-feel. Instead, Swing components are rendered using native Java drawing functions. This results in a consistent look-and-feel regardless of the platform on which the Java application is running. Figure 2.4 illustrates this, figures (a) and (b) show the same AWT interface running on Microsoft Windows and Motif GUI whilst figures (c) and (d) show the Swing implementation of the same interface running on the two GUIs.



Works Order: 113946
Status: Planned
Template:
Exit Date: 28/11/2001

Customer:	Muller Dairies Uk Ltd	Order Type:	ARTWORK & RE
Design:	Muller Indulge Chocolate Lid	Req'd Despatch:	28/11/2001
Product Code:	0	Req'd Customer:	29/11/2001

	Resource	Prior.	Hrs	Date	Status
Scanning:					
Retouch:					
Artwork:	Artwork Wayne	35	1	27/11/2001	Done
Artwork Proof:					
Artwork QC:					
Repro:	Repro	35	1	27/11/2001	Done
Repro Proof:					
Digital QC:					
Film QC:	QC Film	35	.5	27/11/2001	Done
Cromalin:	Cromalin	35	1	28/11/2001	In Progress
Platemaking:					
Wet Proofing:					

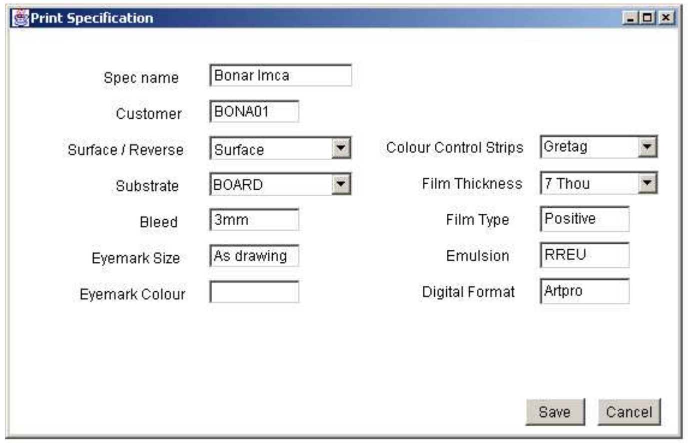
Product:	CROMALIN
Via Despatch:	Yes
Films:	Yes
Comments:	

Record: 30 of 147

Figure 2.3: An example of a Visual Basic form

2.2.2 Structure of form interfaces

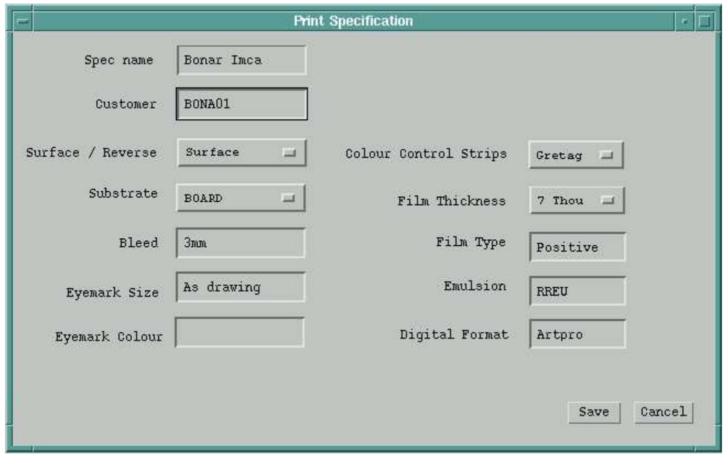
A common method of accessing the forms in a form interface is through the use of menus. These may be implemented as menu forms or as drop down menus. Where a menu form is used buttons on the menu form link to data entry forms which either hide the menu form or simply appear *on top* of it. Closing the data entry form returns the user to the menu form. In complex form interfaces which contain many data entry forms, there may be several levels of menu form which are used to partition the data entry forms into related groups. The form interface in Appendix B uses such an approach to separate forms for maintaining lookup tables from the main data entry forms. Menu forms are a popular method of accessing forms in HTML form interfaces due to the limited set of interaction components and the complexity required to create drop down menus in HTML. Figure 2.5 illustrates an HTML form interface that utilises frames to present the window contents in a *menu* and *main* area. The menu area holds the menu form which is always visible whilst the main area holds the data entry forms that are displayed as a result of selecting options on the menu form. Whilst this example uses a single browser window to display forms it is also possible to launch forms in new browser windows


Print Specification

Spec name	Bonar Imca		
Customer	BONA01		
Surface / Reverse	Surface	Colour Control Strips	Gretag
Substrate	BOARD	Film Thickness	7 Thou
Bleed	3mm	Film Type	Positive
Eyemark Size	As drawing	Emulsion	RREU
Eyemark Colour		Digital Format	Artpro

Save Cancel

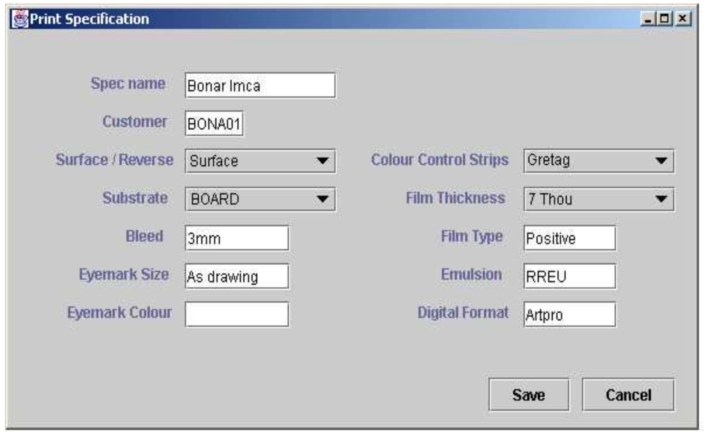
(a) Windows AWT


Print Specification

Spec name	Bonar Imca		
Customer	BONA01		
Surface / Reverse	Surface	Colour Control Strips	Gretag
Substrate	BOARD	Film Thickness	7 Thou
Bleed	3mm	Film Type	Positive
Eyemark Size	As drawing	Emulsion	RREU
Eyemark Colour		Digital Format	Artpro

Save Cancel

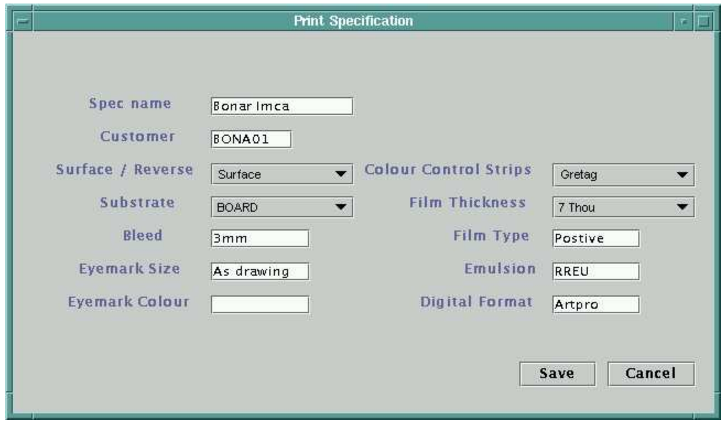
(b) Motif AWT


Print Specification

Spec name	Bonar Imca		
Customer	BONA01		
Surface / Reverse	Surface	Colour Control Strips	Gretag
Substrate	BOARD	Film Thickness	7 Thou
Bleed	3mm	Film Type	Positive
Eyemark Size	As drawing	Emulsion	RREU
Eyemark Colour		Digital Format	Artpro

Save Cancel

(c) Windows Swing

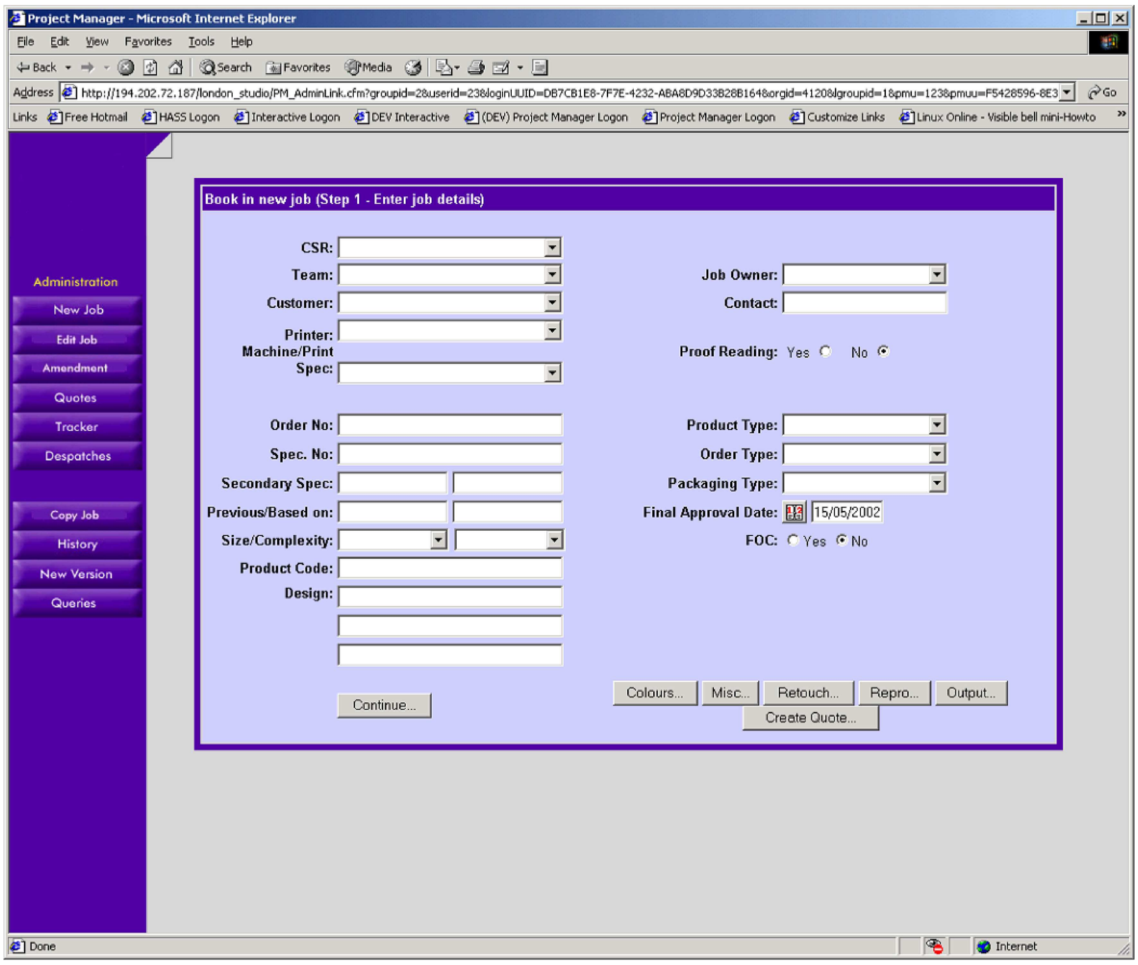

Print Specification

Spec name	Bonar Imca		
Customer	BONA01		
Surface / Reverse	Surface	Colour Control Strips	Gretag
Substrate	BOARD	Film Thickness	7 Thou
Bleed	3mm	Film Type	Postive
Eyemark Size	As drawing	Emulsion	RREU
Eyemark Colour		Digital Format	Artpro

Save Cancel

(d) Motif Swing

Figure 2.4: Difference in look-and-feel among GUI environments


Project Manager - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://194.202.72.187/london_studio/PM_AdminLink.cfm?groupid=2&userid=23&loginUUID=DB7CB1E8-7F7E-4232-ABA8D9D33828B164&orgid=4120&lgroupid=1&pmu=123&pmuu=F5428596-8E3
Links Free Hotmail HASS Logon Interactive Logon DEV Interactive (DEV) Project Manager Logon Project Manager Logon Customize Links Linux Online - Visible bell mini-Howto
Administration
New Job
Edit Job
Amendment
Quotes
Tracker
Despatches
Copy Job
History
New Version
Queries
Book in new job (Step 1 - Enter job details)
CSR:
Team:
Customer:
Printer:
Machine/Print Spec:
Order No:
Spec. No:
Secondary Spec:
Previous/Based on:
Size/Complexity:
Product Code:
Design:
Job Owner:
Contact:
Product Type:
Order Type:
Packaging Type:
Continue...
Colours...
Misc...
Retouch...
Repro...
Output...
Create Quote...
Done
Internet

Proof Reading: Yes [✓ No]

Final Approval Date: 15/05/2002

FOC: Yes [✓ No]

Figure 2.5: Menu in an HTML form interface

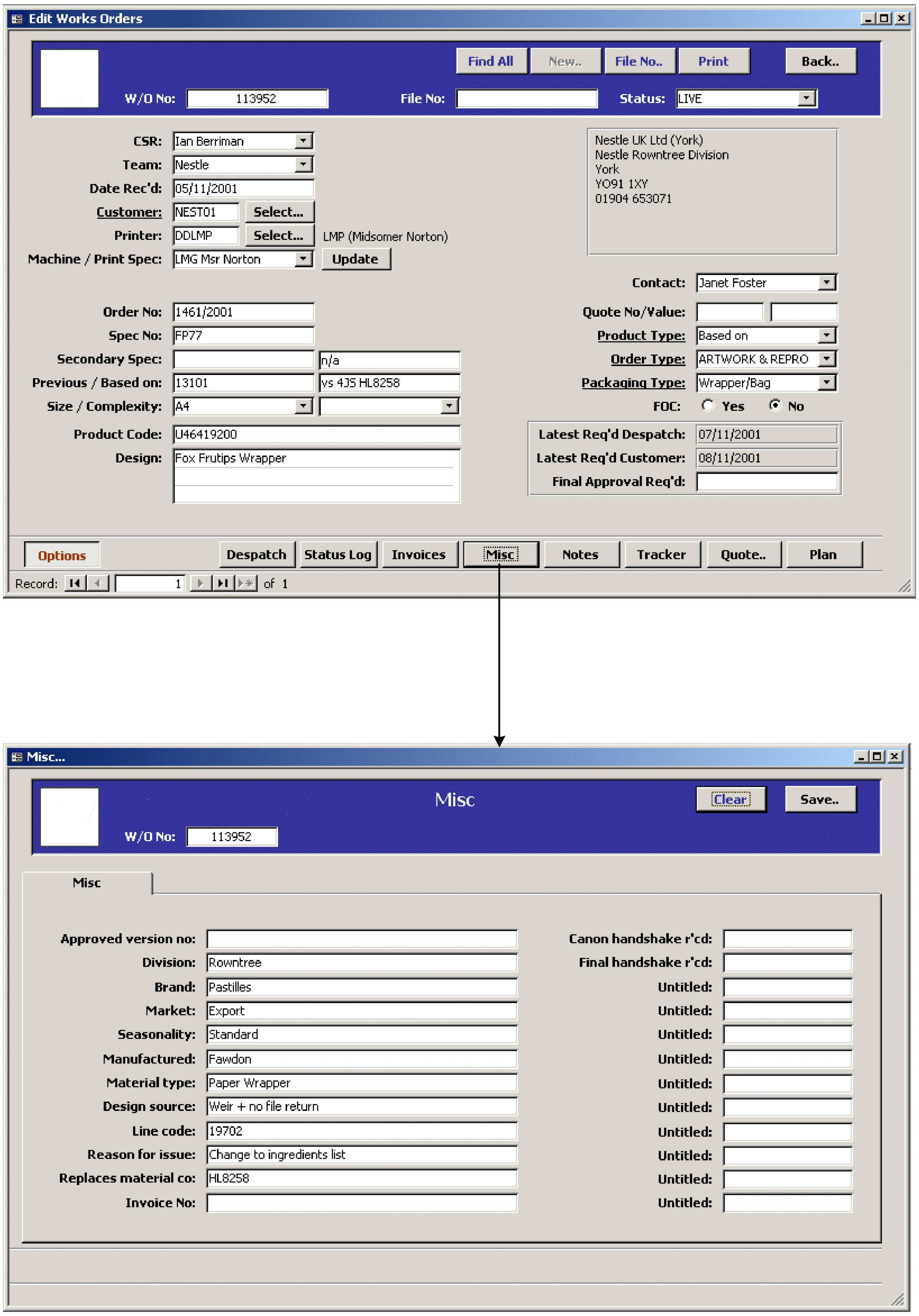
thus replicating the behaviour of Java and Visual Basic form interfaces. The links between HTML forms are implemented as hyperlinks which may be represented visually using a number of paradigms including the conventional text link, buttons or clickable images. In Figure 2.5 the hyperlinks in the menu frame are represented using images.

An alternative to using menu forms as a means of accessing data entry forms is to employ drop-down menus. Where drop-down menus are used, related functions are often placed under the same drop-down menu, just as related functions are placed on the same menu form. It is common for form interfaces that use drop-down menus to employ a multiple document interface (MDI) paradigm. Under this scheme, a main window holds the menu bar and drop-down menus. Forms are represented using child windows that are opened, closed and operated on in response to menu selections made in the main window.

Besides the links between menu and data entry forms, links may also exist among data entry forms. For example, Figure 2.6 illustrates two data entry forms which are linked via buttons. The upper form captures the core information associated with the production of packaging whilst the lower form holds optional information not stored with every job. Hence, the lower form is normally hidden and can be accessed by clicking the Misc button on the upper form. In addition to button links between forms other controls can also be used. Clicking on rows of lists or tables is a popular method, and is often used to open another form which gives a more detailed view of the information contained in the row. Figure 2.7 demonstrates such a link.

Besides clicking on interface controls to launch other forms it is also possible to use many other *events*, triggered as a result of user interaction, to link forms. Combo boxes in Microsoft Windows can be made to respond to the entry of text that does not correspond to an entry in their drop down list, for example. When a new item is entered it is possible to detect this and perform some action, opening a form for example. This kind of link can be used to implement the entry of a new value in a lookup table and the capture of detailed information about the lookup value. For example, a combo box could be used to allow users to select, or enter a new customer code, on an order form. If a new customer code were entered, that was not in the list, then the customer form would be opened to allow the customer's details to be captured.

As a consequence of the links that can be made between forms it is possible to think of form interfaces not merely as a collection of data entry forms but as a hierarchic, and in more complex cases, network-like structuring of forms.



Edit Works Orders

Find All | New.. | File No.. | Print | Back..

W/O No:	113952
File No:	
Status:	LIVE
CSR:	Ian Berriman
Team:	Nestle
Date Rec'd:	05/11/2001
Customer:	NEST01
Printer:	DDLMP — LMP (Midsomer Norton)
Machine / Print Spec:	LMG Msr Norton
Order No:	1461/2001
Spec No:	FP77
Secondary Spec:	n/a
Previous / Based on:	13101 — vs 4JS HL8258
Size / Complexity:	A4
Product Code:	U46419200
Design:	Fox Frutips Wrapper

Nestle UK Ltd (York)
Nestle Rowntree Division
York
YO91 1XY
01904 653071

Contact:	Janet Foster
Quote No/Value:	
Product Type:	Based on
Order Type:	ARTWORK & REPRO
Packaging Type:	Wrapper/Bag
FOC:	Yes [✓ No]
Latest Req'd Despatch:	07/11/2001
Latest Req'd Customer:	08/11/2001
Final Approval Req'd:	

Options | Despatch | Status Log | Invoices | Misc | Notes | Tracker | Quote.. | Plan

Record: 1 of 1

Misc...

Misc

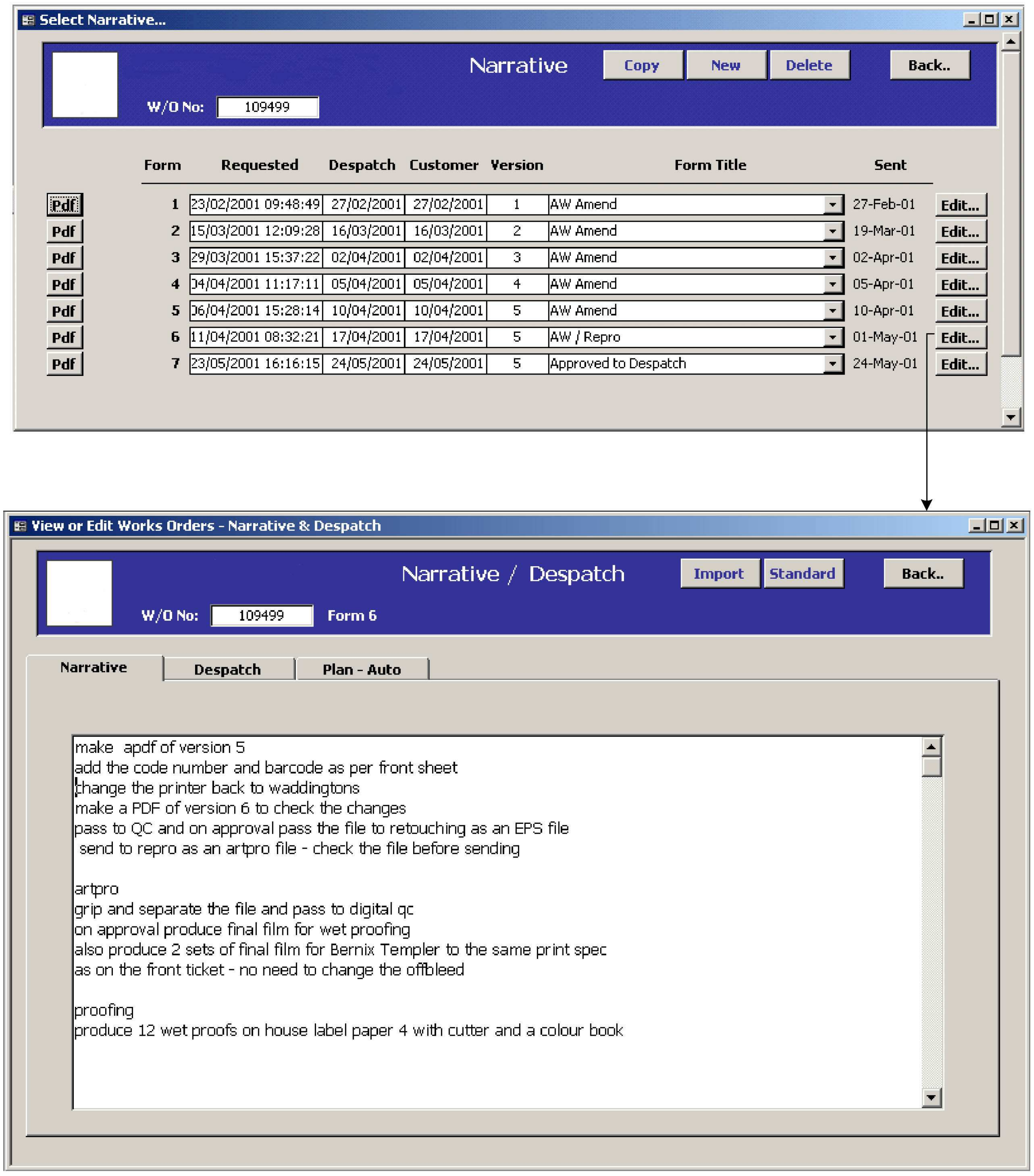
Clear | Save..

W/O No: 113952

Misc

Approved version no:		Canon handshake r'cd:	
Division:	Rowntree	Final handshake r'cd:	
Brand:	Pastilles	Untitled:	
Market:	Export	Untitled:	
Seasonality:	Standard	Untitled:	
Manufactured:	Fawdon	Untitled:	
Material type:	Paper Wrapper	Untitled:	
Design source:	Weir + no file return	Untitled:	
Line code:	19702	Untitled:	
Reason for issue:	Change to ingredients list	Untitled:	
Replaces material co:	HL8258	Untitled:	
Invoice No:		Untitled:	

Figure 2.6: Button link between data entry forms



W/O No: 109499

Form	Requested	Despatch	Customer	Version	Form Title	Sent
1	23/02/2001 09:48:49	27/02/2001	27/02/2001	1	AW Amend	27-Feb-01
2	15/03/2001 12:09:28	16/03/2001	16/03/2001	2	AW Amend	19-Mar-01
3	29/03/2001 15:37:22	02/04/2001	02/04/2001	3	AW Amend	02-Apr-01
4	04/04/2001 11:17:11	05/04/2001	05/04/2001	4	AW Amend	05-Apr-01
5	06/04/2001 15:28:14	10/04/2001	10/04/2001	5	AW Amend	10-Apr-01
6	11/04/2001 08:32:21	17/04/2001	17/04/2001	5	AW / Repro	01-May-01
7	23/05/2001 16:16:15	24/05/2001	24/05/2001	5	Approved to Despatch	24-May-01

W/O No: 109499 Form 6

```
make  apdf of version 5
add the code number and barcode as per front sheet
change the printer back to waddingtons
make a PDF of version 6 to check the changes
pass to QC and on approval pass the file to retouching as an EPS file
 send to repro as an artpro file - check the file before sending

artpro
grip and separate the file and pass to digital qc
on approval produce final film for wet proofing
also produce 2 sets of final film for Bernix Templer to the same print spec
as on the front ticket - no need to change the offbleed

proofing
produce 12 wet proofs on house label paper 4 with cutter and a colour book
```

Figure 2.7: Linking forms via rows

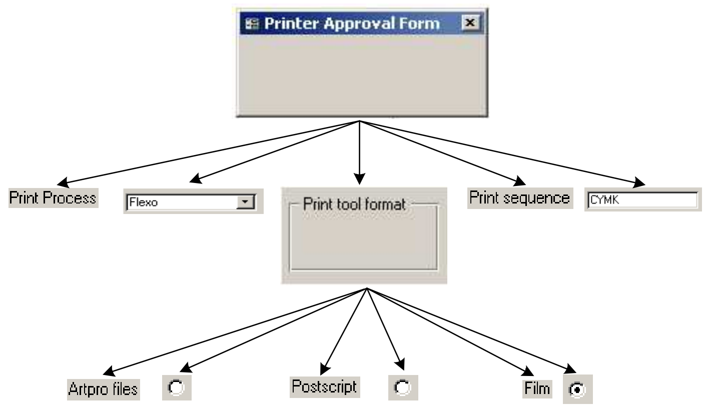


Figure 2.8: Hierarchic representation of a form

2.3 Abstract representations of form interfaces

The previous section introduced the notion of a form interface by examining the components used to build form interfaces and looking at how form interfaces are structured. The diversity in the look-and-feel of the same form under different GUI environments was also illustrated. Here we take a step back from the *toolkit* view, and examine forms at a more general level. Consider the stages of construction of a typical form using direct manipulation such as that found in Visual Basic. Construction begins with a blank window. Interface controls are placed onto this window to represent the form fields. Some interface components are placed within other interface components. Radio buttons within a groupbox, for example. Thus, the form could be thought of as being built up in layers. The lowest layer can be thought of as the form (window) whilst the highest layers can be thought of as form fields and labels represented by interface controls such as textboxes and radio buttons. The intervening layers are made up of controls such as groupboxes. Figure 2.8 shows how the layers are built up. At the root of the tree is the window (form) whilst the leaves of the tree represent the form fields and labels. Internal nodes of the tree represent groups of related form fields. The arcs in the tree represent the *containment* relationship.

Whilst the tree representation of a form demonstrates how it is possible to remove the surface appearance of a form such as the exact positioning of fields it still remains to remove references to the implementation toolkit. Thus, for a form implemented using Java Swing components we must describe the form without reference to the JFrame component that implements it and textboxes without references to JTextField components that implement them. One way to achieve this is to define a set of *generic* controls that are simply described by name and have no visual representation. When combined with the tree view the generic controls then allow a form interface to be described in an abstract fashion. Each generic control will map to several implementations, each one in a separate toolkit. For example, the generic control *Table* would map to the Swing component, JTable, in a Java implementation whilst in an HTML implementation it might map to a structure such as the one shown in Figure 2.2. Similarly, the generic control button could map to both a Windows push button and an HTML hyperlink implemented using an image. Table 2.1 illustrates how a mapping might look for combo boxes and radio buttons. Using generic names we can restate the form described in Figure 2.8 so that it is a completely abstract representation (see Figure 2.9).

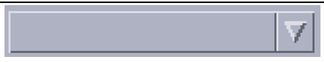
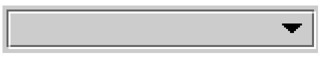
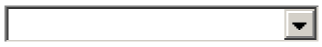
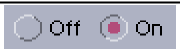
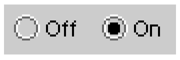
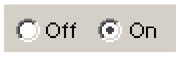
Generic Name	**Toolkit**	**Implementation**
combobox	Motif	
	Swing	
	Windows	
radio button	Motif	
	Swing	
	Windows	

Table 2.1: The mapping from generic control to implementation

Describing interface controls using generic names enables forms to be modelled

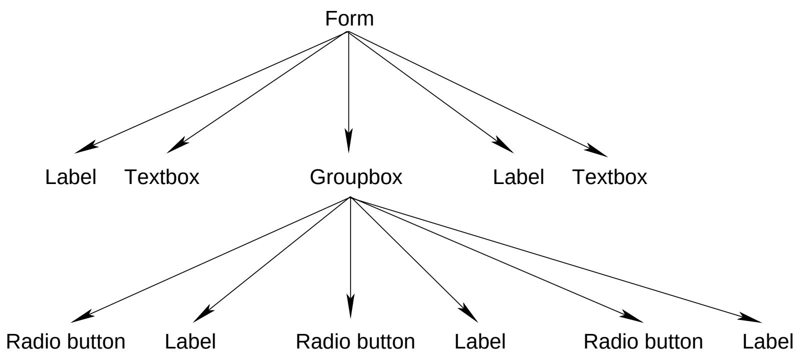


Figure 2.9: Concrete Interface Model for Figure 2.8

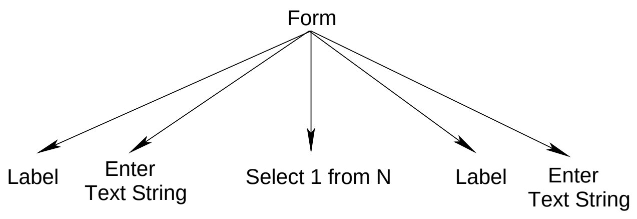


Figure 2.10: A functional description of the concrete interface model in Figure 2.9

irrespective of their implementation, however, the descriptions still resemble those used by toolkits. It is possible, though, to gain a higher level of abstraction by describing fields functionally. For example, a textbox control might be described as “enter a character string” whilst a slider might be described as a “number chooser”. At this level of abstraction it would be possible for a textbox to also be described as “enter a numeric value”. Thus the mapping between functional descriptions and interface components is many-to-many. At such a high level of abstraction it is possible to describe not only form interfaces but also paper-based forms. Figure 2.10 shows how the form in Figure 2.9 can be described functionally.

If individual forms can be represented as rooted trees then a collection of trees could be used to model a form interface comprising of several forms. However, whilst such a representation would capture the structure of the forms in the interface, it could not represent the different ways in which forms are linked together. In graphical form interfaces, for example, the use of buttons to connect forms often results in a network-like structure. Thus for the purpose of modelling graphical form interfaces, directed graphs are a more suitable representation.

2.4 Using abstract representations

The previous section demonstrated how it was possible to describe forms at different levels of abstraction. A abstract representation of forms is an essential requirement of the work presented here. By using a medium level representation such as the one described above it is possible to strip away the surface appearance of the forms in the interface to reveal their underlying structure. We are then free to reason about form interfaces regardless of whether they are implemented using HTML or Java Swing components and irrespective of the graphical user interface environment which is in use. Furthermore, the functional abstraction provides an insight into the equivalence of forms as it illustrates how different interface components can be used to represent the same information. However, such a high level description us not necessarily sufficient for describing form interfaces as it does not capture enough detail.

The abstract representations of forms that we have seen in this chapter act as a starting point from which to build a more comprehensive model of form interfaces. Such abstractions focus only on the representation of individual forms, whereas our focus is form interfaces. It will be necessary therefore to extend the abstractions so that they are able to capture the relationships amongst forms as discussed in section 2.2.

2.5 Summary

This chapter has introduced form interfaces using examples of Java, Visual Basic and HTML implementations of forms. It is demonstrated how forms are represented in graphical user interfaces and how they are linked to create network-like structures. The problem of developing an abstract representation of forms has been discussed along with how such an abstraction representation of form interfaces will act as the basis for the rest of the work described in this dissertation.

Chapter 3

The use of forms in the design and re-engineering of database schemata

3.1 Introduction

The aim of this chapter is to review the literature relating to the use of forms in database analysis and design and in the process of reverse engineering information systems applications. The research is relevant to the work presented in this thesis as it proposes automatic and semi-automatic mechanisms for recovering a conceptual schema from a collection of forms or reports. Many of the approaches discussed here have concentrated upon either paper- or character-based forms or reports and differ from our work which deals with graphical form interfaces.

In order that our review of these approaches be coherent, the approaches are compared on four dimensions. This framework for analysis is presented in Section 3.2 and is then followed by the discussion of the individual approaches. We have classified each approach as belonging to one of two categories: database analysis and design; or database reverse engineering.

3.2 Framework for Analysis

The first dimension relates to the way in which the analysis of forms is undertaken. Two paradigms were found: *area-based* in which forms are partitioned into areas of related fields before being analysed; and *form-based* in which the complete form is analysed. The paradigm used impacts on the complexity of the recovery process. Area-based methods have two phases of integration. The first phase integrates the schemata produced for each area into a form schema followed by the second phase which integrates the form schemata into a single database schema. Form-based methods require only the integration of form schemata into the database schemata.

The second dimension relates to whether a particular approach makes use of an existing database schema from the database that underlies the forms or reports. The use of an existing schema/data dictionary may provide considerable help in recovering a conceptual schema as it provides a rich source of information that can be used during recovery. Tables in a relational database can be used as an indicator of entity-types and their attributes, for example. Consequently, approaches that utilise an existing database schema may require less human intervention.

The third aspect of comparison is the type of conceptual schema that is recovered. This may be an Entity-Relationship or object oriented schema.

The fourth dimension relates to user intervention. The extent of user intervention required by a particular approach depends on: (i) the amount of information that can be obtained from form or report definitions and existing schemata; and (ii) whether or not an approach employs heuristics. In the case of (i), the more information available from the form or report the less the user has to be involved. In the case of (ii) approaches making use of heuristics often present the user with a list of possibilities which a user must confirm or reject. Those approaches not employing heuristics often ask the user to specify missing information.

3.3 Forms used in database analysis and design

For the purpose of comparing forms-based approaches to database analysis and design we differentiate between expert system and non-expert system approaches.

3.3.1 Rule-based Approaches

Whilst the approaches of Choobineh *et al.* (1992) and Holsapple *et al.* (1982) can all be classed as rule-based, they differ considerably in two respects. The first difference relates to the method used to capture the model of the reports or forms that are to be analysed. Holsapple *et al.* (1982) use the notion of a report schema to represent reports. A report schema comprises a set of record types (i.e. relations) and a set of binary relations among record types, which may have a cardinality of 1:1, N:1, 1:N or M:N. Before analysis of reports takes place, a report schema for each report is constructed by examining the reports manually. This contrasts with the approach of Choobineh *et al.* (1992) which employs an expert system to support the capture of the formal definitions of forms known as form types. A form type

	EDDS	**Batini et al**	**Holsapple et al**	**Ridjanovic**	**Mfourga**	**ITOC**
Paradigm	Incremental	Area-based	Form-based	Form-based	Form-based	Form-based
Rule-based	Yes	No	Yes	No	Unknown	Unknown
Recovered Data Model	ER	EER	Network Model	Relational Model	ER	Proprietary
User Intervention	Semi-automatic	Manual	Automatic	Semi-automatic	Manual	Automatic
Information Used	Form definitions	Form definitions	Report definitions	form layout, database schema	form definitions, database schema	source code, form definitions, database schema
Purpose	Database Design	Database Design	Database Design	Database Design	Reverse Engineering	Reverse Engineering

Table 3.1: Comparison of approaches to form analysis

defines the structure, constraints and presentation of the fields on a form. Fields are classed as being either atomic or grouped, the latter consisting of a set of constituent atomic fields. For each field, the following information is recorded: *datatype* which denotes the set of values that can be entered in the field; *origin* which indicates the source of the field (e.g. entered by user, calculated, copied from another field/form); *cardinality* which denotes the number of possible values a field can accept; and *level* and *node* which are used to capture the hierarchic structure of forms.

Capturing form types is achieved using a form definition system (FDS) that enables forms to be sketched using a full screen editor. Included in the FDS is an inference component that supports both novice and expert modes of operation. Through a combination of heuristics and interaction with the user, the inference component constructs form types from the forms sketched by the user. The construction of a form type follows several stages: hierarchy inference; node key inference; and dependency inference. Hierarchy inference attempts to extract the hierarchical structure of a form by clustering fields in nodes based on the adjacency. By examining whether fields can accept null or duplicate values, node key inference aims to identify fields in a node that can be used functionally to determine the other fields in the node. Finally, dependency inference aims to infer functional dependencies among fields in different nodes by examining node keys and the hierarchic structuring obtained during hierarchy inference. In addition to using heuristics to analyse the form sketches, the FDS also seeks user input. In novice mode, user input is restricted to providing example “filled-in” forms and confirming/rejecting suggestions made by the inference component. In the expert mode of operation the user is asked explicitly to supply requested information such as node keys.

The second difference between the approaches of Holsapple *et al.* and Choobineh *et al.* is the knowledge encoded in their rules and how their rules are applied. The focus of the rules in (Holsapple *et al.* 1982) is the transformation of report schemata into an extended network data model via an intermediate representation known as binary report schemata (BRS). Furthermore, Holsapple *et al.* aim to generate a report schema that is optimal in the sense that only the minimum information necessary is recorded in the schema. Thus the rules encode: (i) mappings from report schemata to BRS; (ii) knowledge of restructuring operations on BRS; and (iii) a set of transformations from BRS to extended network model schemata. The three groups of rules correspond to the three stages of extraction: report analysis; schema restructuring; and schema creation, respectively. Extraction is a completely automatic process requiring no input from the user. This contrasts with the expert database design system (EDDS) (Choobineh *et al.* 1992). The knowledge encoded in the rules of EDDS is that required to recognise occurances of entity- and relationship-types and attributes from a set of form types. Furthermore, an element of fuzziness is built into the rules in the form of weightings. Where the information available to the rules is insufficient to make a decision as to what structure has been identified, EDDS will consult the user, either for additional information or ask them to confirm or reject a particular assertion. In this respect EDDS differs from the approach of Holsapple *et al.* (1982) as it employs human intervention rather than applying an algorithmic transformation from the forms or reports to the data model.

The use of human intervention in EDDS, therefore, makes it a semi-automated approach to extraction. The rules in EDDS are divided into six sets: form selection; entity identification; attribute attachment; relationship identification; cardinality identification; and consistency checks. The form selection rules are used to choose a form for analysis to which the remaining five sets of rules are applied. Whilst forms are selected individually for analysis, the analysis does not result in an individual ER schema being created for each form. Rather, as each form is analysed the ER structures identified are added to an ER schema that evolves incrementally as the analysis of forms progresses. One benefit of this approach is that it frees the user from having to deal with the integration of several ER schemata, an operation which has to be performed manually in other forms-based approaches and which requires a significant amount of familiarity with the data model being used. Of the remaining five stages, four deal with the mapping of form structures to ER constructs whilst the fifth stage is concerned with checking the consistency of the evolving ER schema. The first stage of extraction, entity-identification examines node keys, field origins and dependencies among fields in an attempt to identify those forms structures that act as entity-types in an ER schema. For example, a form field that is the source of another form field or the determinant of a field may represent an entity-type. Field names with common endings such as NUMBER, NO., or # may indicate the primary key of an entity-type.

Having identified the entity-types present on a form, stage two attempts to attach attributes to the entity-types. Examples of rules used in this stage include those based on the proximity of fields and on dependencies among fields. If a set of fields are within a predetermined proximity of a field that has been recognised as an entity-type then the fields are designated as attributes of that entity. Similarly, if the left-hand side of a field dependency has been identified as an entity-type then fields appearing on the right-hand side of the dependency are taken to be attributes of that entity-type.

Stages three and four of extraction aim to identify relationships among entity-types and their cardinalities. The hierarchic structure of the form and field dependencies are used during the application of these rules. Hierarchic structures often represent 1:N relationships among entity-types, for example. During relationship and cardinality identification the user is asked to supply role-names for the relationships and provide more information in the event of ambiguity.

When the analysis of a form is complete the consistency checking stage ensures that all the fields on a form appear in the ER schema and that a relationship exists among the ER constructs representing adjacent levels in the form's hierarchy.

Finally, the form selection rules are invoked again to select the next form. This process continues until all the forms have been analysed.

3.3.2 Other Approaches

Those database design approaches not employing a rule-based approach also exhibit a number of differences. Most significant is that Batini, Demo, and Di Leva (1984) present a completely manual approach whilst Ridjanovic (1992) presents a semi-automated approach. Furthermore, the approach described by Batini *et al.* (1984) is area-based whilst the approach described by Ridjanovic (1992) is form-based.

The manual approach of Batini *et al.* sets it apart from most of the other approaches discussed in this chapter, which, whilst constructing the formal representation of forms/reports manually, perform analysis either automatically or semi-automatically. The rationale behind the choice of a manual approach for both model construction and analysis is based on the observation that forms are so diverse and complex in structure that it is not possible to perform analysis automatically. To enable the analysis of forms a framework by which forms can be classified is employed. Forms are decomposed into four parts: a *certificating* part containing identification numbers and dates; an *extensional* part that is "filled-in" by the user; an *intensional* part that contain field names and labels; and a *descriptive* part that contains instructions on how the form should be completed. The main focus of analysis are the intensional and extensional parts of the form which are strictly interconnected and thus analysed together (Batini *et al.* 1984). In addition, the descriptive part of a form is also examined as it often provides a source of constraint information.

The process of extracting the EER schema for a collection of forms proceeds in four stages: form analysis; area design; form design; and interschema integration. Form analysis corresponds to the manual phases of the other approaches as its aim is to create the models of forms that will be used as input to the extraction process. During form analysis the parts of a form are identified and the extensional/intensional parts are decomposed into *areas* of related fields. Areas are central to the mapping from form structures to EER structures, and their properties (e.g. field names, synonyms) are recorded in a *glossary* for each form.

Having constructed a glossary for each form, the area design stage extracts an EER schema for each area of a form. Realising the EER schema for an area uses instances of "filled-in" forms in addition to information obtained from the form's glossary. The exact details of the mapping are contingent on the user who is following the methodology and thus assumes the user is familiar with database design and analysis using the Entity-Relationship approach. Area design results in several EER schemata each corresponding to an area of a form. These are known as *area schemata*. Contrast this with form-based approaches in which the result of form analysis is a schema for the whole form. It may be assumed that the added granularity of form description introduced by the area-design stage reflects the complex nature of forms as perceived by Batini *et al.*. The remaining two stages of the extraction process deal with the integration of schemata. Area schemata are integrated to produce form schemata which are subsequently integrated to produce an application schema that will support the collection of analysed forms. Both integration stages require the identification and resolution of several types of conflict (see Kashyap and Sheth (1996) for a classification of schema conflicts) and requires

Form type	Areas
description	description
list	list
parent-child	description + list
parent two children	description + two lists
grandparent-parent-child	two description + list

Table 3.2: Ridjanovic's form types

a user who is familiar with Entity-Relationship modelling to perform this task. This suggests the target audience of this approach are database developers. Furthermore, the restructuring of schemata which also takes place during integration is very much an informal activity and may explain why the task of form analysis is difficult to automate.

Due to the manual nature of Batini *et al.*'s approach, it can be seen more as a methodology, than a system for extracting a conceptual schema from a collection of forms such as that described by Ridjanovic (1992). Ridjanovic's approach is based on the assumption that forms share the same underlying data model as the database to which they provide access and that the forms and database schema can be mapped via this shared representation into each other. This assumption is used by Ridjanovic to define a set of form types. The type of a form is based on the areas from which it is made up. An area is a collection of fields structured in a particular manner; *description areas* comprise a set of non-repeating fields; and *list areas* comprise a table structure of repeating fields. Table 3.2 lists the form types that are defined by Ridjanovic.

The extraction of a schema from a collection of screen forms is trivial in that each area of a form is mapped to a possibly unnormalised relational table. Certain types of form (e.g. parent-child) correspond to large structures comprising several tables and relationships. A parent-child form type corresponds to a one-to-many relationship between two tables, for example. Information on the primary keys of tables and the cardinality/optionality of certain relationships is obtained through consultation with the user of the system. The result of mapping forms in this way is a collection of unnormalised relational schemata, one per form. Ridjanovic refers to these as *screen subschemas*. Each screen subschema comprises several relations, one for each area on the corresponding form. Often these relations will be unnormalised and a process of normalisation is applied to each "area relation" resulting in the decomposition of the area relation into several other relations, linked by relationships. During this phase it would appear as that an existing data model is used to aid in the normalisation process. Ridjanovic's approach has a number of similarities to that of Batini *et al.*, especially in the use of complex schemas to describe each area of a form. The approaches differ, however, in the time at which the complex area schemata are created. Batini *et al.* create their schema during the analysis phase whilst Ridjanovic creates the schema after analysis has taken place.

3.4 Forms used in database reverse engineering

The use of forms in the reverse engineering of information systems applications has been the focus of both theoretical and commercial research efforts. In Harrison and Lim (1998) a system for reverse engineering an INGRES application by forms (ABF) application into an Oracle Forms application is described. This project has been driven by the commercial need for tools to convert legacy systems into new software development environments. Mfourga (1997) proposes a framework for describing data entry forms and extracting Entity-Relationship schemata from them. The focus of Mfourga's approach, however, is more theoretical investigation than an attempt to develop a commercial system.

The reverse engineering of database schemata from collections of forms is based on the assumption that forms impose a structure upon communications messages (Tsichritzis 1982) and that this structure closely resembles the structure of the underlying database. By analysing the structure of the forms, therefore, an initial database schema can be produced which is later subjected to a process of refinement.

In the framework proposed by Mfourga, forms are represented using a form type. Each form has a layout (i.e. how it appears in the screen or on paper) and is composed of one or more structural units, each comprising several fields. The fields within a structural unit are described in terms of an underlying data source (i.e. database table) and a linked attribute (i.e. an attribute in the database table). Fields whose values are computed have no underlying data source or linked attribute. Structural units are related via either a 1:1 or 1:N relationship which results in a hierarchy of structural units being created for each form. Constraints are also associated with a form type.

A collection of several form types constitutes a form model schema and is created by a process of static and dynamic analysis carried out by the user. Static analysis results in identification of the form layout, structural units and the relationships between structural units, whilst dynamic analysis examines instances of forms to identify constraints.

Using the form model schema and the underlying database schema, a six stage process is used to extract an Entity-Relationship schema for each form. Table 3.3 gives a brief overview of each stage and its purpose.

Whilst Mfourga's approach is described as semi-automatic, a significant amount of user involvement is required, in particular during the construction of the form schema which is a manual process. This contrasts with the forms definition system described in Choobineh, Mannino, and Tseng (1992) which supports the user in creating a

Entity derivation	Translates each structural unit into an entity
Relationship derivation	Translates each relationship between two structural units into a relationship between two entities
Attribute attachment	The fields in a structural unit become attributes of the entity represented by the structural unit
Cardinality determination	The cardinalities of the relationships between structural units are used as the cardinalities of the relationships in the ER schema
Normalisation	Informal procedure of restructuring
Schema Integration	The ER schemata for the forms are integrated by hand into the ER schemata for the forms application

Table 3.3: Mfourga's extraction process

similar form schema using an expert system approach. The ER extraction process also places a considerable amount of emphasis on the relational database schema underlying the forms. Indeed, it is questionable to what extent form information is used during extraction.

The INGRES to Oracle conversion (ITOC) project represents commercial interest in the re-engineering of forms-based applications. The ITOC design recovery tool which has been developed collaboratively by the Centre for Software Maintenance at the University of Queensland and Oracle Corporation enables INGRES ABF applications to be converted into Oracle Developer 2000 applications. The ITOC tool employs data flow analysis techniques to analyse data entry forms and queries. By analysing forms, the columns to which form fields are related can be identified and ,through the analysis of queries, relationships among data underlying the forms can be identified.

The conversion process comprises an analysis stage and a generation stage. Generation is handled using existing Oracle CASE tools once the required application model has been created in the analysis stage. Analysis of ABF applications follows several stages and utilises form definitions, 4GL source code and the underlying relational database schema. Central to the conversion process is data flow analysis in which links are established between tables in the database and queries appearing in the 4GL code. Query analysis uses the relationships identified by data flow analysis to identify relationships among the tables that appear in queries in the 4GL code. Form analysis aims to associate each form field to a single attribute in the database. Having done this forms analysis schemata can be created. Each INGRES form is composed of blocks of related fields called frames. These are mapped onto Oracle table usage specifications which are subsets of the attributes of tables. The relationships among the table usage specifications are also determined at this stage.

The ITOC approach differs considerably from that taken by Mfourga in both its aims and its complexity. Firstly, ITOC aims for a complete end-to-end re-engineering of an application whereas Mfourga's approach is limited to creating a conceptual schema that could support the set of analysed forms. Secondly, the use of source code analysis results in a more complex extraction procedure than that employed by Mfourga.

3.5 Summary

The analysis of the information conveyed by a collection of forms has been the subject of research attention from both a database design and database reverse engineering perspective. The majority of the approaches have concentrated upon analysing paper- or character-based forms, and there is a sharp contrast among the different methods of analysis. Batini *et al.* (1984) advocate a manual approach due to the diversity and complexity of forms that could be encountered. Despite this, several other approaches have proposed successful automatic or semi-automatic methods of analysis. For describing the information conveyed by forms the Entity-Relationship model has been employed. The only method concentrating upon graphical form interfaces is that of Ridjanovic (1992), however, it does not take into account the individual controls that may appear in a graphical form interface, instead choosing a high-level approach decomposing forms in areas containing several fields.

Whilst the approaches reviewed here have provided some insight into existing approaches to capturing the information conveyed by forms it has also highlighted a deficiency in the understanding of how interface structures in graphical form interfaces are represented.

Chapter 4

Graphical Form Interfaces

4.1 Introduction

Due to the diverse range of GUI environments, all of which employ slightly different methods of presenting what is conceptually the same interface component, it is important to represent form interfaces in a manner that is independent of any particular GUI environment. The aim of this chapter is to expand on the previous chapter by examining potential methods for representing forms at an abstract level, the ultimate aim being to define a model that can be used to represent form interfaces regardless of their GUI environment and implementation language.

The chapter begins by examining various interface presentation models to understand how they could be used to model form interfaces. User interface design guidelines relevant to form interfaces are then discussed and used to identify the class of form interfaces that will be focussed on in our work. The form interface model is then introduced, first informally, using examples to highlight how it is able to capture the information necessary to reason about form interfaces. This is followed by formal definition of the model.

4.2 Interface Presentation Models

An interface presentation model is a high-level representation of the interaction objects used in a graphical interface and the relationships between them. Gray *et al.* (1998) identified two levels of presentation model: the user interface toolkit and the abstract interaction model.

4.2.1 User Interface Toolkits

Chapter 2 presented a number of example forms from different GUI environments and also illustrated interface controls available in popular user interface (UI) toolkits. Interface controls are graphical objects that can be used to represent the attributes and/or actions of other objects (Galitz 1997). User interface toolkits are libraries of interface controls from which GUIs can be created. As the examples in Chapter 2 demonstrated UI toolkits may target a single platform such as the Microsoft Windows toolkit, or it may target several platforms, such as the Java AWT and Swing UI toolkits. Whilst not strictly a toolkit, HTML offers the ability to construct user interfaces than can be used on several platforms. Furthermore, its textual nature provides a method of describing form interfaces without reference to the appearance of controls and, as identified in Chapter 2 this is a key requirement for an abstract representation of form interfaces. The drawback of HTML is the limited set of controls available. However, initiatives such as the XForms language (W3C 2002) represent a step towards support for richer set of interface controls.

4.2.2 Abstract Interface Models

User interface development environments (UIDEs) have been developed to simplify the task of specifying and designing user interfaces. Operating at a higher level of abstraction than the user interface toolkit UIDEs employ several models to capture: the tasks the interface is to support; the data that will be operated upon; the interaction objects; and the structure of the interface. The model representing the interaction objects and the structure of the interface is often called an abstract interface model as it enables representation of interfaces independently of their look-and-feel. Abstract presentation models usually comprise two main components: a set of interaction objects; and a hierarchic representation used to capture the relationships between interaction objects.

4.2.2.1 Interaction Objects

Interaction objects can be modelled at three levels of abstraction. Figure 4.1 illustrates the relationships between each level. The highest level of abstraction is the abstract interaction object (AIO). At this level it is the purpose of the interaction that is significant, not the details of specific controls (Wilson, Johnson, Kelly, Cunningham, and Markopoulos 1993). AIOs, therefore, describe the type of interaction required, entering a number, for example. The JADE (Judgment-based Automatic Dialog Editor) tool which forms part of the GARNET system (Myers, Giuse, Dannenberg, Vander Zanden, Kosbie, Pervin, Mickish, and Marchel 1990) defines a set of seven AIOs. These include: *single-choice*, allowing the selection of

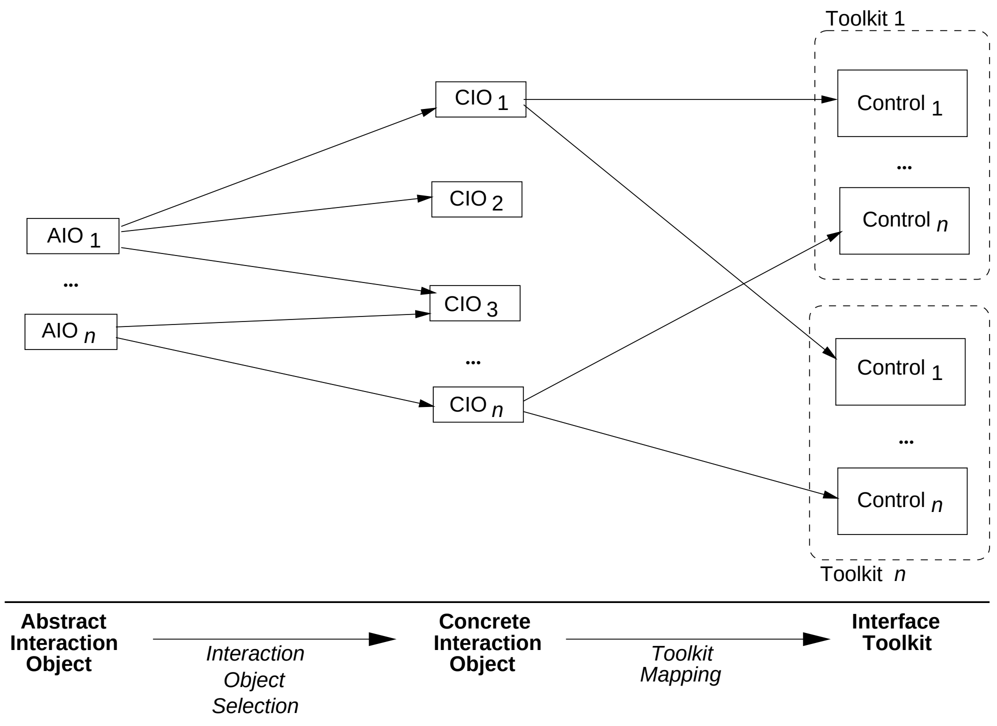


Figure 4.1: The levels of interaction object

one item from a group of several; *multiple-choice*, allowing the selection of one or more items from a group of several; *text*, allowing text strings to be entered; *single-choice with text*, allowing either text entry or selection of one of several options; *multiple-choice with text*, allowing text strings to be entered or selection of one or more options; *command* allowing the selection of items from a menu; and *number-in-range*, allowing the selection of a number within a given range. The Teallach model-based interface development environment (Griffiths *et al.* 1999) uses a more general AIO representation and defines several broad categories of interaction technique:

- *Chooser* - allows items to be chosen from a list;
- *Display* - allows data to be displayed;
- *Editor* - allows data to be changed; and
- *ActionItem* - allows actions to be invoked

The Teallach categories subsume JADE's interaction techniques. The *chooser* category, for example, subsumes the single-and multiple-choice techniques; the *editor* category the text interaction technique; and the *ActionItem* the command technique. Furthermore, the Teallach interaction techniques are not limited to the pre-defined categories. If necessary, user-defined categories can be added.

A data-centred approach to specifying AIOs is used in ADEPT (Wilson *et al.* 1993). In this approach the AIO is described in terms of a data-type. An interaction object that is used to represent a person's age, for example, would be described as a number, whilst their surname would be described as text. The TRIDENT interface development environment (Bodart, Hennebert, Leheureux, Provot, Sacre, and Vanderdonckt 1990) also uses a data-centred approach by classifying interaction objects according to the type of data they are able to represent.

Concrete interaction objects (CIOs), despite their name, are not implementation specific. Instead CIOs are represented using generic names for controls. The name "textbox", for example, is used to refer to a control that allows a text string to be entered. A many-to-many mapping exists between abstract and concrete interaction objects. That is, a single AIO can be represented by several concrete interaction objects and the same concrete interaction object can be used to represent several AIOs. Consider an AIO for capturing a numeric value, for example. This could be represented by a textbox or a slider. The textbox might also be used for an AIO that allowed input of a text string hence the M:N mapping. The exact mapping between AIO and CIO for a particular interface depends on a number of factors including the type of data and the level of user experience. Indeed, a significant amount of research has been directed at selecting appropriate interaction objects (see, for example Bodart and Vanderdonckt (1994); and de Barar, Foley, and Mullet (1992)).

The lowest level of interaction object are the controls themselves implemented as toolkits. A one-to-many mapping exists between concrete interaction objects and interface controls. A textbox, for example, maps to its Motif implementation in the Motif toolkit, the Micorosft implementation in the Microsoft Windows toolkit and so on. This level of description, therefore, corresponds to that of the toolkit described in Section 4.2.1. Some approaches blur the relationship between concrete interaction objects and controls. Teallach, for example, employs the Java Swing set of interface objects as CIOs. By doing this, different interface styles are supported because the Swing components have pre-defined mappings to the Motif, Microsoft Windows and native Swing toolkits. JADE removes the CIO level completely through the use of "pluggable" interface style libraries which enable the AIO to map directly to a control in the current selected interface style.

4.2.2.2 Relationships among interaction objects

As well as possessing the ability to represent controls an abstract interface model must also be able to capture the relationships between interaction objects. This enables the structure of windows and dialog boxes to be specified at a high level. The relationships among interaction objects are captured using trees, the root of the tree being the window interaction object. The leaves of the trees represent either AIOs or CIOs depending on the level of specification, whilst the internal nodes are used to specify related groups of interaction objects. The arcs of the tree capture the containment relationship between controls. The benefit of modelling window and dialog box specifications as trees is that presentation details such as colours, fonts and the exact position of widgets are removed. This is important as the colour depth and font capabilities may vary among windowing environments. Furthermore, different windowing environments have different layout styles for controls such as push-buttons. In Microsoft Windows (Microsoft 1995), for example, push-buttons are positioned horizontally along the bottom edge of windows whilst Apple style guidelines (Tognazzini 1985) suggest buttons are positioned vertically down the right-hand side of windows. By not specifying the exact position of controls the window specification remains independent of any particular windowing environment.

4.2.3 Discussion

We have chosen the concrete interface model as the basis for our model of form interfaces for two reasons: CIOs provide a means of representing controls that are independent of GUI environment thus removing the difference in look-and-feel among different GUI platforms as discussed in the previous chapter; and the mapping from CIO to toolkit control in a given GUI is sufficiently close so as to avoid ambiguity. If we had chosen to use AIOs then too much information would have been lost, making it impossible to map an interface captured in say, HTML, to Visual Basic as we would not know which toolkit controls to use when reconstructing the interface in Visual Basic. Such information is also essential, as will be shown in Chapter 7, for comparing the information content of form interfaces.

Some extensions to concrete interface models will also be necessary to extend certain CIOs to enable them to capture sufficient information to allow the comparison of information. In particular, complex CIOs such as grids, lists and combo boxes, all of which may have complex internal structures. Consider, for example, a grid which could contain several columns and might display data in its rows that could be classified as being of one or more types. To capture this extra information we introduce two new CIOs: the row and the column. A row may contain two or more columns and a listbox or grid may contain one or more rows (see Figure 4.2 for an exact specification). Similarly, the list of a combo box may contain two or more columns.

A second modification we make to concrete interface models is to unify the "label" CIO with the other *interactive* CIOs, such as textboxes. Doing this simplifies the description of forms as the CIO representing the form field and the CIO representing the field label are treated as single object, rather than separately as they are normally in concrete interface models. This would seem a sensible approach, given that many interface builders include a label automatically when placing controls on forms. Labels can also provide clues as to the contents of controls. Rock-Evans (1989), for example, describes a data analysis technique in which the presence of plural labels is taken to signify a repeating group, with the repeating entity being determined by the singular version of the plural label.

4.3 Guidelines for constructing form interfaces

There are very few restrictions placed upon the use of interface controls and it is up to interface designers to follow *best practice* guidelines to prevent poorly designed interfaces. Research studies such as Brown (1998), Mayhew (1992) and Smith and Mosier (1986) have all proposed such guidelines that suggest the circumstances in which a control should be used; its orientation; the positioning of its associated label; and whether any grouping should be performed. Some guidelines also give the type of the data that a control is best suited to representing. Table 4.1, summarized from Galitz (1997), shows some common guidelines for controls used in form interfaces. Many interface tools and generators use guidelines when generating interfaces. The model-based system TRIDENT uses guidelines to select controls based on data from a data model, whilst the forms editor of Microsoft's Visual Basic automatically

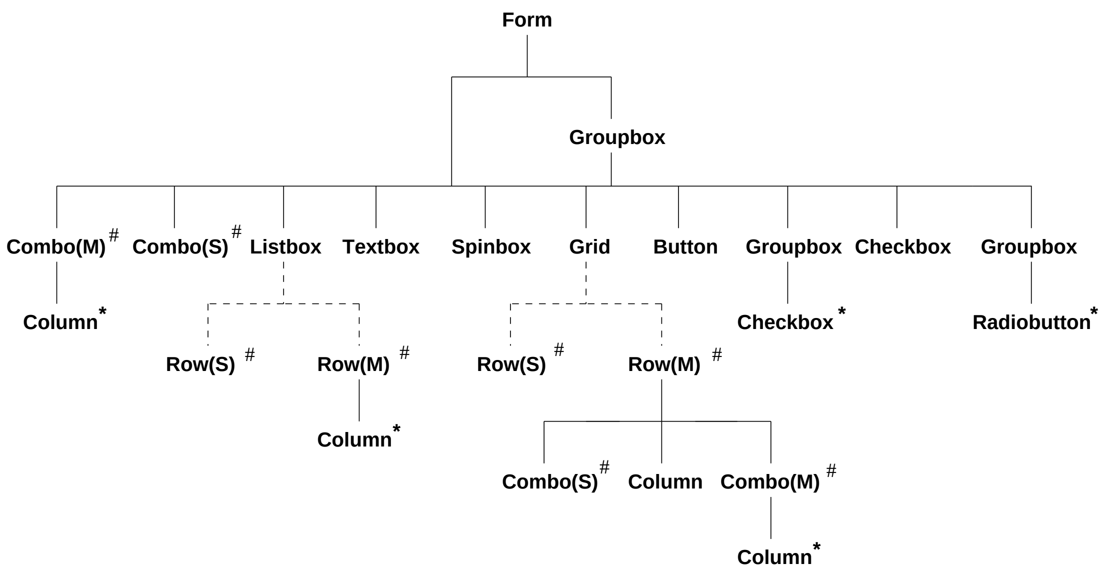


Figure 4.2: A model of the concrete interaction objects found in a graphical form

groups radio buttons when they are placed on a form, for example. Interface guidelines are potentially useful for our work in that they define a smaller more manageable set of possible interfaces that can be compared using the techniques we develop.

Whilst user interface design guidelines represent *best practice* advice on how forms should be constructed in a graphical user interface, it is at the designers' discretion whether or not they choose to follow them. There are, however, rules of interface construction that cannot be violated. Such rules pertain to the manner in which interfaces controls can be combined to construct an interface. To capture these *structural* constraints for graphical form interfaces, we introduce the concept of a general model of the CIOs found on a form. This provides a schema against which instances of graphical forms can be validated. The model also serves to define the class of forms that are dealt with in this work. Figure 4.2 shows a graphical representation of the model. The nodes correspond to fields represented by concrete interaction objects; where a CIO is followed by an (S) or (M) this indicates the single- or multi-column variant of the CIO. An asterisk beside a node indicates that

Control	**Label Position**	**Usage Guidelines**
Textbox	Above Left	Permit entry, editing and display of alphanumeric data. Used where the length of data varies (e.g. names) and domain of possible values is too large for list selection. Single line mode for data not exceeding one line. Multi-line mode for data spanning several lines (e.g. addresses)
Combo box	Above Left	Allows alphanumeric data to be entered/displayed, as well as option to choose a value from a list. Used for selecting a single choice from a frequently changing list of values where the list has more than 9 items. List-s may have 1 or more columns although only the first column is shown in the unextended state.
Spinbox	Above Left	Allows a single alphanumeric value to be selected from a list of possible choices. Used where the list of choices changes infrequently and is consecutively ordered and short in length (e.g. days of week).
Listbox	Above Left of first row	Permits lists of tables of alphanumeric information to be display. Used for lists which change frequently and which contain more than 9 items. Items may be selected by clicking their row. Lists may have several columns.
Grid	Above Left of first row	Permits the entry and display of lists and tables of alphanumeric values which are subject to frequent change. Used for lists/tables that require entry functionality and which have more than 9 items. Rows may be selected by clicking them. A grid may have several columns.
Groupbox	Top-left	Used to group together related controls visually in a titled frame.
Radio buttons	Right of each radio button	Permit the selection of a single value from a small set (between 2 and 8) of choices which are not subject to change. Used for discrete data which can be described textually. The group of radio buttons are enclosed in a groupbox and should be arranged in a vertical (preferred) or horizontal orientation.
Checkboxes	Right of each checkbox	Allow the selection of 1 or more values from a small set of choices (between 2 and 8) which do not change and where a meaningful textual description of each choice can be given. The group of checkboxes is preferrably arranged in a vertical orientation, otherwise it is arranged horizontally, and a groupbox is used to enclose the checkboxes
Checkbox	Right of each checkbox	Permit a simple boolean value to be set/unset.
Button	Centered within the button	Used to initiate actions (e.g. display a window).
Window (form)	Title bar	Acts as a container on which other controls can be placed.

Table 4.1: GUI Form design guidelines

this control must appear two or more times for each instance of its parent. This enforces the constraint that multi-column variants of a CIO must appear have at least two columns and that groups of CIOs require at least to CIOs to appear in the group. Nodes labelled with a # signify that they are able to act as links to other forms (see Section 4.3.1). Arcs in the model represent *contains* relationships among fields. Where an arc between a CIO and its children is represented by a dashed line this signifies that only one of the children may appear as a child of the parent. This prevents the mixing of single and multi-column variants of a CIO under the same parent. In addition to the structural constraints the model is also able to encode the design guidelines which suggest that radio buttons and checkboxes, when used in groups, should be enclosed in a groupbox.

4.3.1 Linking forms

The previous chapter identified several methods of linking forms together. These ranged from simple button clicks to the entry of text in a combo box. It is possible to define two broad classes of link between forms. Clickable links, that are initiated by the clicking, or double clicking of an interface control; and text entry links, that are triggered as a result of typing in a text field such as a combo box. Regardless of the type of link the end result is always the opening of another form. So, we refer to the relationships between forms as *opens* relationships.

Using controls such as buttons, rows and combo boxes, form interfaces can be created that are network-like in structure. In the next section we will exploit the graph-like properties of form interfaces implemented in a GUI environment to develop a graph-based representation of them.

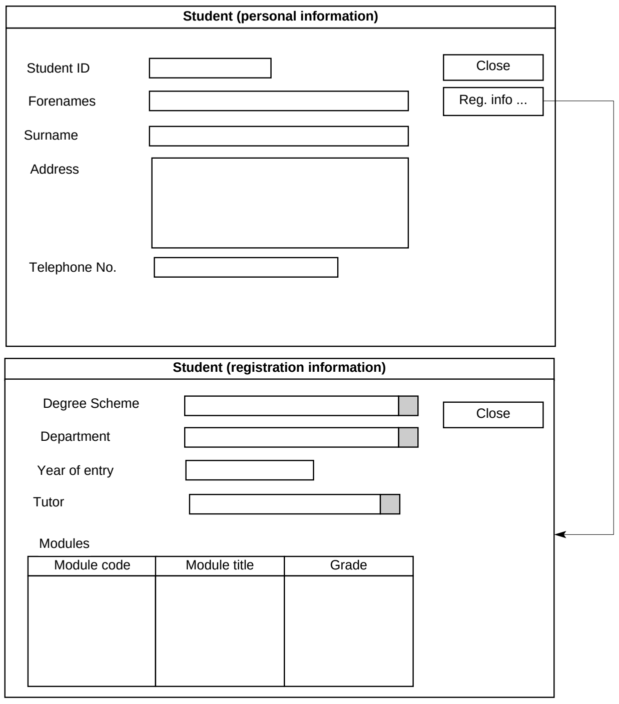


Figure 4.3: Form interface representing student information

4.4 Form Interface Model

4.4.1 Modelling instances of form interfaces

In this section, a graph-based model of form interfaces is developed. The model employs concrete interaction objects to represent interface controls and generalises the hierarchic presentation models used in UIDEs to represent window and dialogue box definitions.

Before defining the form interface model formally, we shall present an example of how the model can be used to represent a form interface and discuss the concepts of the model informally. Consider the two forms shown in Figure 4.3 that represent an interface for recording details about university students. Figure 4.4 shows an instance of the forms model representation of an interface, this is called a forms interface schema (FIS).

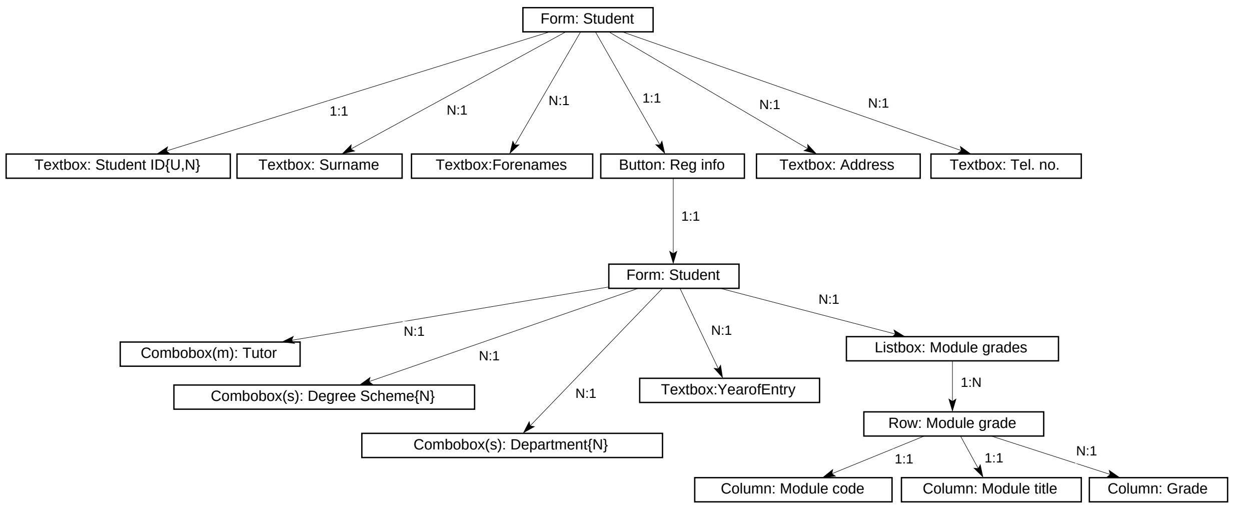


Figure 4.4: Example form interface represented using the form interface model

Nodes in the FIS represent the fields of a form. Depending on the field several properties are recorded, these are: the type of control; the name of the object represented by the field; the label used to refer to the field; the domain of the object; and any constraints on the field. The control type of a field is the name of the CIO used in the field (e.g. combo box). Where a CIO has single and multi-column variants a letter in parentheses (e.g. combobox(m)) is used to signify the particular variation.

The name of a field is distinct from the label that appears with the field in the interface. The name refers to the object represented by the field and can be thought of as a unique identifier of a data source that gives the field its value. It is possible for several fields to have the same name and this signifies that these fields represent the same object as they obtain their values from the same data source. In Figure 4.3 the Module code field appears on both forms, for example. A field's label is used as the legend of the field in the interface. Thus two fields with the same data source can have different labels.

Two constraints are supported on fields: *not null* which means that a value must be entered in the field; and *uniqueness* which means that a particular value can be entered only once in a field for all instances of a given form. Constraints are represented on the forms schema in braces after the field name, N indicates a not null constraint and U a uniqueness constraint.

The properties that a field possesses are determined by the position of the field's CIO in the general model of a form. For those CIOs appearing at leaf positions all the properties are defined, however, for fields appearing internally only the name and label fields are defined. The remaining properties of these nodes are dependent on their children and are realised through the aggregation of the properties of their child nodes.

Edges in the forms schema represent either *contains* relationships or *opens* relationships. For a contains edge the tail indicates the container and the head the contained field. The tail of an opens edge represents the field that responds to user interaction and the head represents the form that is opened in response to the event. Edges also possess a cardinality which is similar to that in the Entity-Relationship model. In the form schema an edge cardinality refers to the number of instances of the concept, represented by the tail of the edge, there can be in relation to the concept represented by the head of the edge. Take, for example, the Student form and the Surname field which are connected by an edge of cardinaility N:1. This means that several instances of the member form (i.e. several students) can have the same value in the Surname field. Cardinalities of 1:1, 1:N, N:1 and M:N are allowed, however, there are some restrictions on how they are assigned to edges. For example, it is impossible to assign a cardinality of 1:N to an edge which has a field that can only accept a single value (e.g. textbox) at the N-end of the relationship.

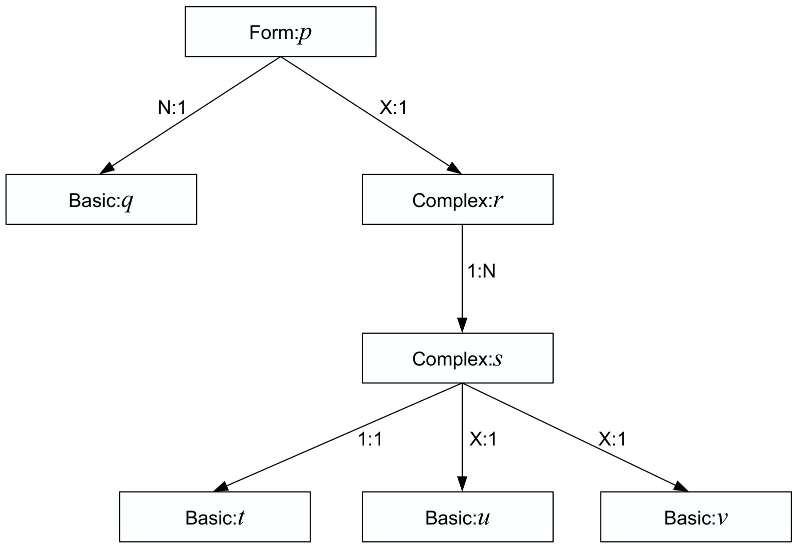
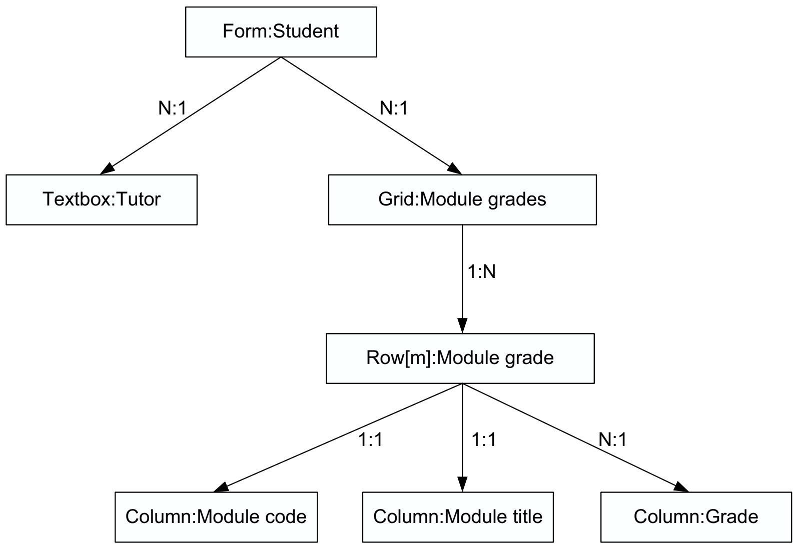


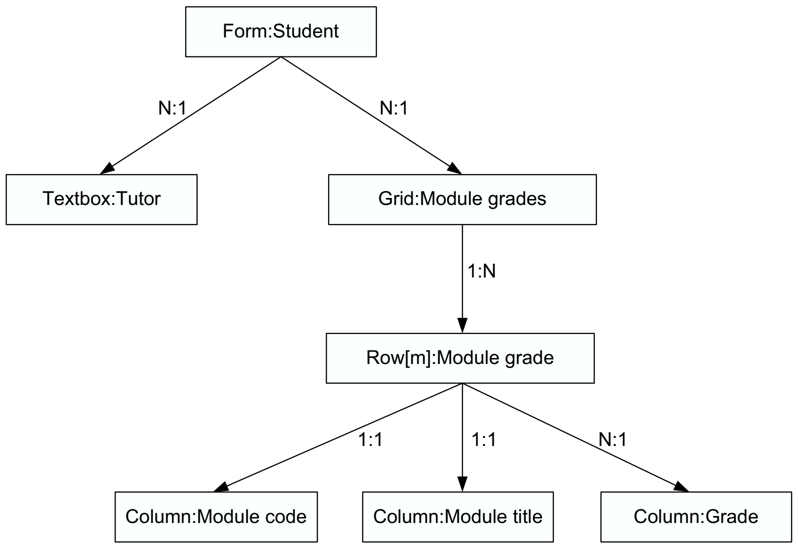
Figure 4.5: A simple pattern

4.4.2 Representing Patterns

A second requirement of the form interface model is the ability to represent forms at two levels of abstraction. The previous section demonstrated the lower of the two levels. This section examines the higher level abstraction, that is necessary to support the graph-based pattern matching system developed in Chapter 6. In the context of forms, a pattern is like a generalisation of several forms. Consider the pattern shown in Figure 4.5, for example. The letters in italics represent "variables" to which concrete values may be bound, whilst the identifiers Complex and Basic refer to sets of interface controls. The cardinalities 1:1, and N:1 have the same meaning as in the previous section whilst the cardinality X:1 means that X can be either 1 or N. A pattern, therefore, describes a class of form structures. If we take the pattern in Figure 4.5 and assume that $\mathsf{Complex} = \{Grid, Listbox, Row[m]\}$ and $\mathsf{Basic} = \{Textbox, Column\}$ then two members of the class of this pattern are shown in Figure 4.6. By looking for the members of a pattern's class in a form interface it is possible to identify particular structures in that form interface. Figure 4.6(a) demonstrates this by identifying the student form containing the listbox in Figure 4.4. This process of locating structures is called pattern matching.



(a)



(b)

Figure 4.6: Members of the class of the pattern in Figure 4.5

When performing pattern matching, it is sometimes necessary to specify conditions that will govern the match. Consider again the form interface in Figure 4.4 and the pattern in Figure 4.5. Let us assume that we want the pattern to match the student form, only when there is no textbox on the form called "Year of entry". Such a condition requires the ability to recognise the presence and, absence of structures.

In order to support patterns, the form interface model must provide:

- *variables* - patterns must be applicable to many instances of form interfaces and it is impossible to know the names that will be used for interface controls *a priori*. By using variables as node names it is possible to bind literal values to the variables during matching. This enables a single rule to be applied to any number of form interfaces;
- *wildcards* - When matching patterns it is possible that several, similar, form interface structures map to the same information structure, differing only in the type of controls used and in the cardinality of the controls with respect to their parents. Wildcards allow patterns to match multiple structures; and
- *the absence of structures* - to detect certain information structures in a form interface it is necessary to ensure that particular structures are absent. The form interface model must, therefore, allow patterns to capture this condition.

4.4.3 Form Interface Model

Having discussed our model of graphical form interfaces and its main concepts informally we now define the model formally.

Definition 4.1 (Forms Interface Model) *Let:* Σ_{O} *be the finite set of object names and* Σ_{V} *be the set of symbols used to represent variables. Let* T *be the finite* *set of control types;* T_b *the set of basic control types; and* $T_c = T - T_b$ *be the set of complex (container) control types. Let* T_w *be the finite set of wildcard control types;* T_{wb} *the set of basic wildcard control types; and* $T_{wc} = T_w - T_{wb}$ *be the set of complex wildcard control types. Let* DOM *be the set of domains;* $C = \{1{:}1,\ 1{:}n,\ n{:}1,\ m{:}n,\ x{:}1,\ 1{:}x,\ x{:}n,\ x{:}x\}$ *be the set of edge cardinalities;* $\textsc{Cons} = \{N, U\}$ *be the set of constraints on nodes; and* L *the set of words used to label interface controls. If* x *is a set then the convention* $\mathcal{P}(x)$ *is used to represent the powerset of* x*.*

A forms interface schema is a directed graph $G = (N, E, \lambda, \tau, l, \psi, dom)$ *where:*

- N *is the set of fields and is the union of the sets* $N_{\text{basic}} = \{n : \tau(n) \in T_{\text{b}} \cup T_{\text{wb}}\}$ *and* $N_{\text{complex}} = \{n : n \in T_{\text{c}} \cup T_{\text{wc}}\}$*;*
- N *is partitioned into* N_+ *and* N_- *representing positive and negative nodes respectively*
- $E = E_+ \cup E_-$ *is the set of edges such that* $E \subset \mathcal{E}$ *where* $\mathcal{E} = (N \times C \times \{opens, contains\} \times N)$ *and if an edge* $e = (tail, card, type, head) \in E$ *then:*
  - $card \in \{1{:}1, n{:}1\}$ *if* $\tau(head) \in T_{\text{b}}$
  - $card = \{1{:}1\}$ *if* $\tau(head) = button$
  - $card = \{1{:}1\}$ *if* $\psi(head) = U$
  - $card = \{1{:}n\}$ *if* $\tau(tail) \in \{listbox, grid, checkboxes\}$
  - $card \in C$ *for any other* e
- $\lambda : N \to \Sigma_{\text{O}} \cup \Sigma_{\text{V}}$ *is a function maps names and variable symbols to nodes;*
- $\tau : N \to T \cup T_{\text{w}}$ *is a function that maps types to nodes;*
- $l : N \to L$ *is a function mapping fields to control labels;*

- $\psi : N_{\text{basic}} \rightarrow \mathcal{P}(\textsc{Cons})$ *is the function that maps a basic field to a set of constraints; and*
- *the function* $dom : N_{\text{basic}} \rightarrow DOM$ *is a function that maps from a basic field to its domain.*

$\square$

The forms interface model provides us with a representation of graphical form interfaces that extends the hierarchic presentation models with the ability to model networks of forms. The model also captures those properties of the form fields that are required for the comparison of the information content of graphical form interfaces. We limit our model to a subset of the interface controls found in GUI environments. This subset comprises interface controls that are commonly used in form interfaces and found in almost all modern windowing environments. A similar restriction has been placed on the constraints that are expressible in our model. Rather, than attempt to capture all possible types of constraint that may occur between form fields we have chosen to represent only the subset that are necessary for the comparison of information content.

4.5 Summary

The aim of this chapter was to identify a model suitable for the representation of graphical form interfaces that was independent of any particular windowing environment. A graph-based model, which extends the hierarchic presentation models used in MB-IDEs to include links between forms, has been proposed for this purpose. To ensure that the model is independent from a particular windowing environment concrete interaction objects (CIOs) are used to represent interface controls and where current CIOs proved insufficient they have been extended.

Whilst the model described in this chapter is able to capture form interfaces it is not sufficient to capture the information structures present in form interfaces. In the next chapter we explore how we might represent the information structures present in form interfaces through the use of conceptual data models.

Chapter 5

Form Information Models

5.1 Introduction

The aim of this Chapter is to identify a data model that is suitable for representing the information conveyed by graphical form interfaces. We concentrate on three models: the Entity-Relationship model (Chen 1976); the IFO model (Abiteboul and Hull 1987); and the nested sequence of tuples (NST) model (Guting, Zicari, and Choy 1989). All of these models have been used previously for representing the information conveyed by paper or electronic forms. Furthermore, equivalence metrics or restructuring mechanisms are also available for all three models. Both of the above attributes are desirable as it may be easier to use, or extend, a model that has already been for paper or character-based forms than it would be to adapt a model which had not. The availability of restructuring mechanisms and equivalence metrics is a benefit as it allows us focus on the equivalence properties of form interfaces without first having to develop equivalence metrics for the data model.

Having identified the three models, we describe a tool, based on the work described in (Rollinson and Roberts 1998), that was developed to investigate how graphical form interfaces convey information. Using this tool allowed us to gain an appreciation of the type of information structures likely to occur in form interfaces and this was used to guide the model selection process. The Chapter then goes on to review the suitability of each of the models. Finally, a formal definition of the chosen data model is given.

5.2 Models for representing information on forms

5.2.1 Hierarchic Models

The nested sequence of tuples model was proposed for the purpose of modelling office documents such as forms and reports. The model also includes an algebra, that can be used to manipulate and restructure schemata expressed in the NST model. This allows the creation of new forms through the combination of existing forms, for example. In this chapter, we concentrate on the representational abilities of the model, the algebra will be examined in more detail in Chapter 7 when approaches to restructuring and comparing forms are considered. Central to the NST model are four constructs: atomic attributes, multi-valued attributes, aggregations and repeating groups. At a formal level aggregations and repeating groups are represented using sequences, with aggregations always being limited to sequences of one element. A fundamental feature of the NST model is its use of sequences, which

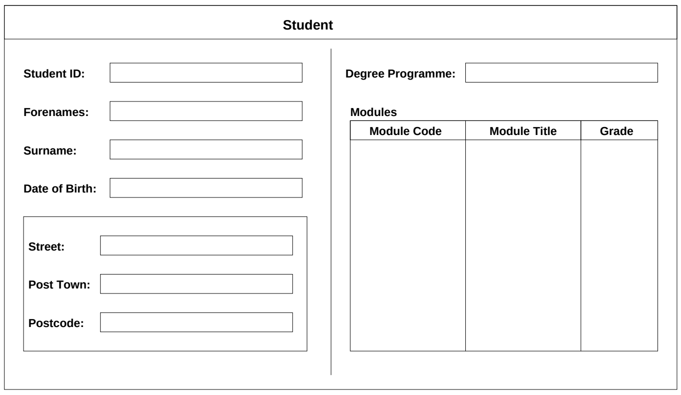


Figure 5.1: A form for recording student details

impose an order on their elements, rather than sets that are used in other hierarchic approaches such as IFO. The rationale behind the use of sequences is to reflect more faithfully the structure of office documents.

Consider the student form in Figure 5.1. Using the NST schema we can capture the information conveyed by this form as illustrated in Figure 5.2. In the diagrammatic representation of an NST schema, circular nodes represent repeating groups and multi-valued fields whilst square nodes represent atomic attributes and aggregations. The diagrammatic representation of the NST model is somewhat peculiar in that it employs a form of shorthand that omits either the tuple or the atomic attribute that is repeated in a repeating group. If this shorthand representation were used in In Figure 5.2 then we would remove the “Module” aggregation and connect the “Module code”, “Module title” and “Grade” directly to the “Modules” repeating group.

The second hierarchic model, the IFO model, was proposed for the purpose of mathematical investigation into the fundamental principles of semantic data models (Abiteboul and Hull 1987). It was later used in an investigation of the representation and restructuring of information in office forms (Abiteboul and Hull 1988). Here we consider the model as discussed in (Abiteboul and Hull 1988) which includes an

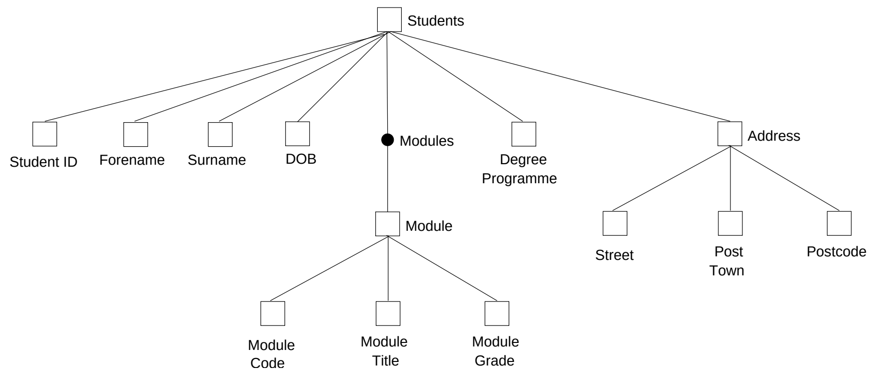


Figure 5.2: An NST schema of the information conveyed by the form in Figure 5.1

additional construct, the single element type, as well as the atomic type, aggregation, generalisation and grouping constructs. The single element type is an atomic type whose domain comprises a single value.

Whilst the IFO and NST models share the same core set of constructs, the two models are quite different, in particular, in the way they represent repeating groups. The IFO model uses sets rather than sequences and in this respect is unable to represent the ordering of documents in the same way the NST model can. The two models also differ in their graphical notations. The IFO notation uses distinct symbols for repeating groups and aggregations and also represents the elements of repeating groups explicitly. Contrast this with the NST model that uses just two symbols to represent all four of its constructs. Figure 5.3 shows the IFO schema for the hypothetical student form in Figure 5.1. Aggregations are represented using the $\otimes$ symbol; groupings with the $\circledast$ symbol; generalisations with the $\oplus$ symbol; and atomic types with the $\square$ symbol. Single element types are represented as 1_x where x is the element of the domain. For the purpose of modelling data entry forms, the union and single element type constructs provide a convenient mechanism for expressing optionality on form fields. In Figure 5.3, for example, the generalisation type *registered on* is able to capture the fact that a student does not have to be registered on a degree programme.

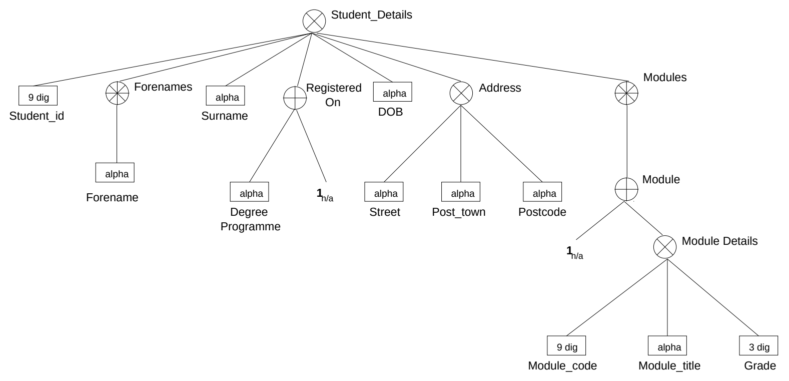


Figure 5.3: IFO schema employing the generalisation and single-valued constucts

5.2.2 Modelling graphical form interfaces using hierarchic data models

A significant benefit of employing a hierarchic data model as a means of representing the data structures underlying a form is the close correspondence between the CIOs on the form and the constructs used in heirarchic data models. This correspondence is illustrated in Table 5.1 adapted from Rollinson and Roberts (1998) that shows the correspondences between CIOs and the three of the constructs from the IFO model. The correspondence is, in many cases, unambiguous. CIOs such as rows and combo boxes, however, require a more complex mapping and it is first necessary to differentiate between single- and multi-column variants of these CIOs. Having done this, it is then possible to map multi-column rows and combo boxes to aggregations without further ambiguity. In order to determine the exact mapping of a single-column row or combo box the entire interface must be examined for forms containing

IFO	**Aggregation**	**Atomic**	**Set-valued**
CIO	form groupbox row combobox[m] textbox	textbox column combobox[s] radiobuttons checkbox	listbox grid checkboxes

Table 5.1: CIOs as hierarchic types

other fields with the same underlying data source as the combo box or row. If one is found then it is likely that the combo box row is acting in a similar manner to a foreign key in a relational database management system, except in this case two forms rather than tables as being linked. Consider two forms: student and degree programme. The student form contains a single-column combo box which has an underlying data source of programme-name. On the degree programme form is a textbox also with a data source of programme-name. Assuming that degree programme names are unique then the combo box is used to establish a relationship between the two forms and is in effect representing the aggregration degree programme. Therefore, to determine the exact mapping of a single column row or combo box, other forms in the interface must be examined for fields with the same data source. We refer to this process as examining a CIO's context in the sense that we are examining other controls in the interface to gain an appreciation of the context in which a control has been employed (i.e. to represent a relationship between two entity-types, for example). We distinguish between two types of context. The global context of a control refers to the complete interface, and any control whose mapping requires the examination of its global context is termed a *global context mapping.* The local context of a control refers to the form on which the control resides, and a control whose mapping requires the examination of its local context is termed a *local context mapping.*

5.2.3 Entity-Relationship Models

In contrast with the two hierarchic models, the Entity-Relationship model has not been used directly for the representation of the information conveyed by forms. It has, however, been used in both the automated creation of forms from data models (see for example Janssen, Weisbecker, and Ziegler (1993)) as well as in approaches to database design (see for example Choobineh, Mannino, and Tseng (1992)). Furthermore, the use of the ER model to represent the information conveyed by forms has largely concentrated upon modelling collections of forms, rather than individual forms. In this respect, the ER model is better suited than hierarchic models as the relationships between forms are not always expressible using hierarchic models as will be shown in section 5.4. One drawback of using the ER model, however, is that the construction of an ER representation of a form is more difficult than the construction of a hierarchic representation, due primarily to the non-heirarchic nature of the ER model.

Figure 5.4 illustrates the ER representation of the student form student shown in Figure 5.1. In the ER model, the hierarchic containment relationship translates to either a relationship-type among entity-types or a relationship between an entity-type and an attribute. The cardinality of the relationships and their participation constraints are determined by examining the type of interface control used in the field and the constraints that have been defined on a field. Suppose a combo box is embedded on a form, for example. The combo box can display at any one time a single value selected from its list. If we treat the form as an entity-type and the combo box as another entity-type the embedding relationship between the form and the combo box translates to a relationship-type among two entity-types. Furthermore, the combo box can only hold a single value, thus the degree of the relationship-type is either 1:1 or N:1. If the form contained a listbox, which can hold many values, then we would expect the relationship to be 1:N or M:N. If $x : y$ is the degree of a relationship-type the type of control gives the y value of the degree. To obtain the x value the instances of a form must be examined. In the combo box example if several forms had the same value for the combo box then we can deduce that the relationship-type has a degree of N:1. If, however, each instance of the form has a different value and the same value cannot appear on two different forms then we may deduce than the relationship-type has a degree of 1:1.

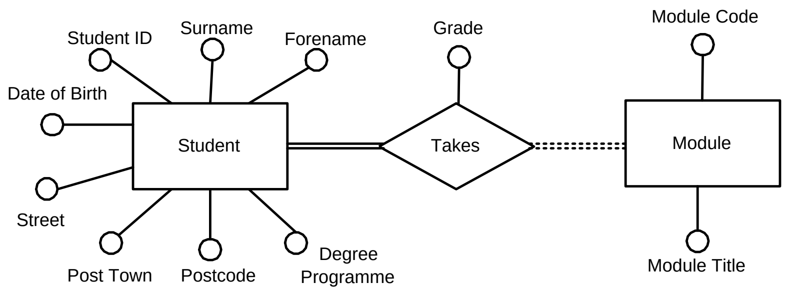


Figure 5.4: Entity-Relaionship schema of data underlying the form in Figure 5.1

The participation constraints of a relationship-type can be found by examining the constraints attached to controls. If a combo box has a “not null” constraint attached to it, for example, and it represents an N:1 relationship-type with the form at the N end of the relationship-type then the N end has mandatory participation.

The lack of a direct correspondence between interface controls and ER constructs also places more emphasis on a control's context. In classifying controls as IFO types the context was used to determine to which IFO type a control belonged. In mapping to ER constructs a context is the minimum unit of mapping because of the lack of 1:1 correspondence. Consider a single-column listbox which maps to a set valued attribute. The listbox maps to the IFO type grouping and the single-column row to the printable type. In the ER model the listbox represents a multi-valued function between an entity-type and an attribute.

5.3 Automatic extraction of information content

To investigate the suitability of data models for the purpose of representing the information conveyed by form interfaces a tool was implemented based on the work reported in Rollinson and Roberts (1998), which describes a rule-based mechanism of capturing the information conveyed by a form interfaces. Forms are represented using the *User interface model*, an abstract model of form interfaces similar to that described in Chapter 4, and the information conveyed by the forms is represented using the Generalised Semantic Model (Hull and King 1987). The rules are implemented in Prolog. A novel aspect of the system is its ability to accept sketches of the UI model as input which makes the process of inputting UI models simple. Sketches are made using the X11 drawing tool Xfig as it uses a simple text-based file format for which details are freely available. The output of the extraction process is also made available in graphical form so the results can be visualised easily. Figure 5.5 shows the architecture of the extraction tool. The area enclosed in the dashed box denotes the core of the system written by the author, whilst the parts of the system falling outside the box use the work of others.

5.3.1 Capturing the UI model

Before the extraction process can begin, a UI model sketch must be transformed into an equivalent Prolog representation. This is achieved using a process known as visual scanning (de Graaf 1996). During visual scanning spatial relationships are extracted from objects in a sketch and described in a spatial relationship graph (SRG). Figure 5.6 shows a UI model instance and its corresponding SRG. By examining the SRG for particular structures (i.e. the UI model primitives) it is possible to arrive at a Prolog description of the sketch. Figure 5.7 shows how the UI model primitives "control" and "relationship" look in a sketch and how they appear at the SRG level. Identifying the structures is a relatively simple process because the SRG is described using a text file that contains a GRL (Graph Representation Language (Manke and Paulisch 1991)) description of the SRG. A Perl script takes the GRL specification and converts it into a collection of Prolog predicates. An example of this conversion for a portion of the SRG shown in Figure 5.6(b) is shown below.

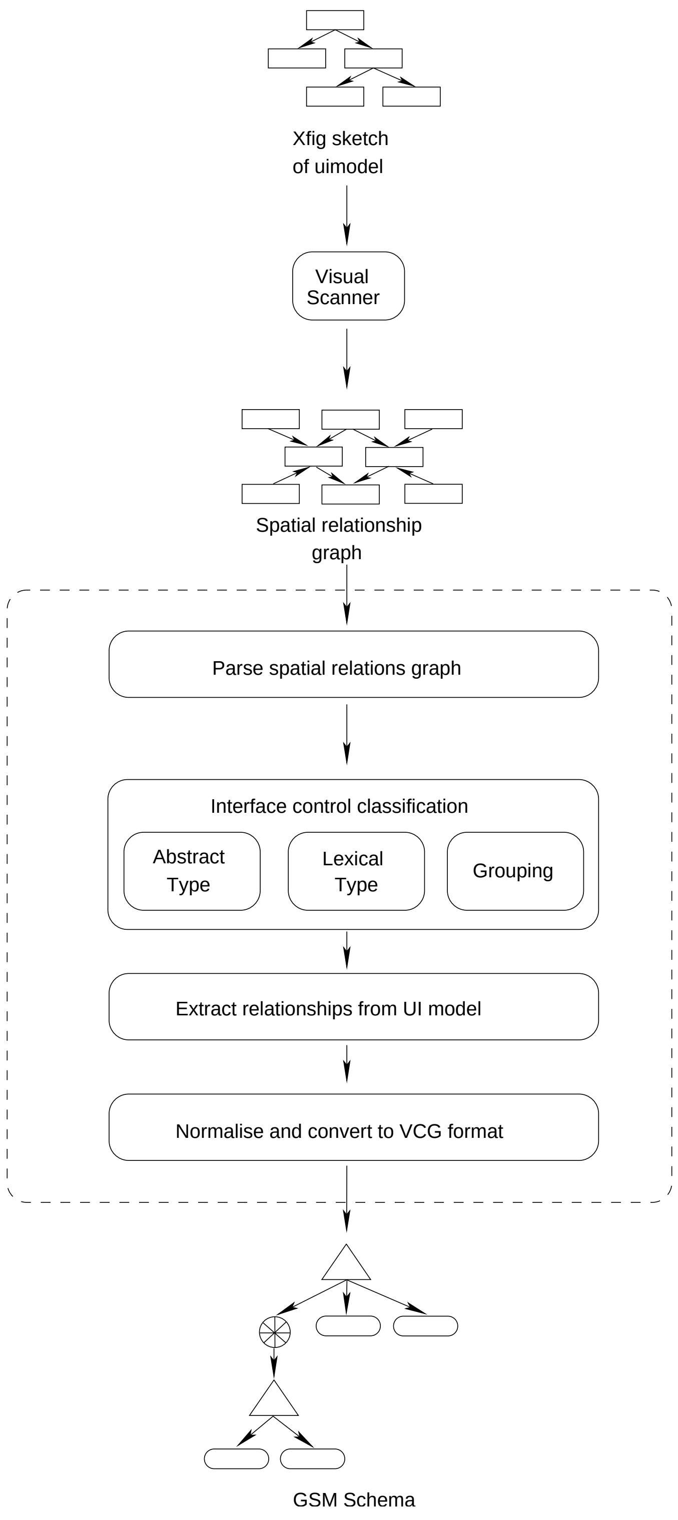
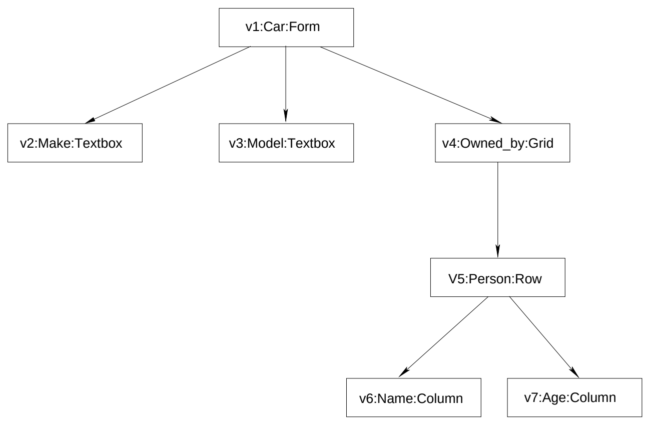
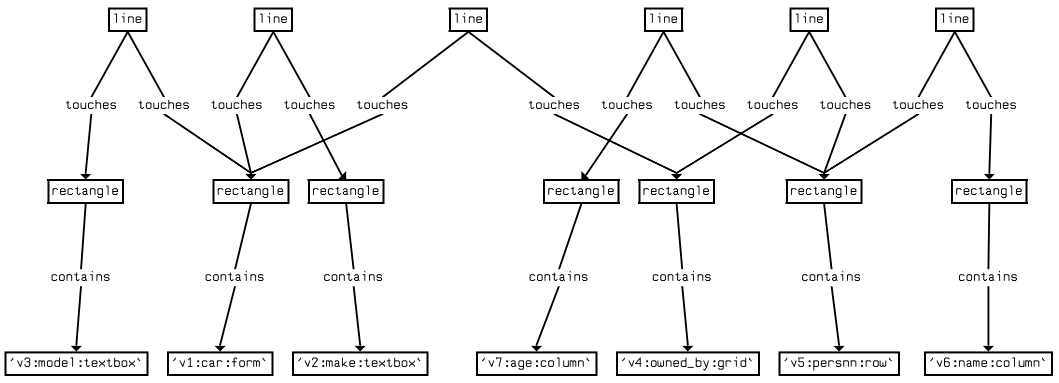

Xfig sketch
of uimodel
Visual
Scanner
Spatial relationship
graph
Parse spatial relations graph
Interface control classification
Abstract
Type
Lexical
Type
Grouping
Extract relationships from UI model
Normalise and convert to VCG format
GSM Schema

Figure 5.5: Architecture of the Prolog mapping system



(a) Xfig sketch of UI model



(b) SRG for the UI model

Figure 5.6: A UI model and its corresponding spatial relations graph

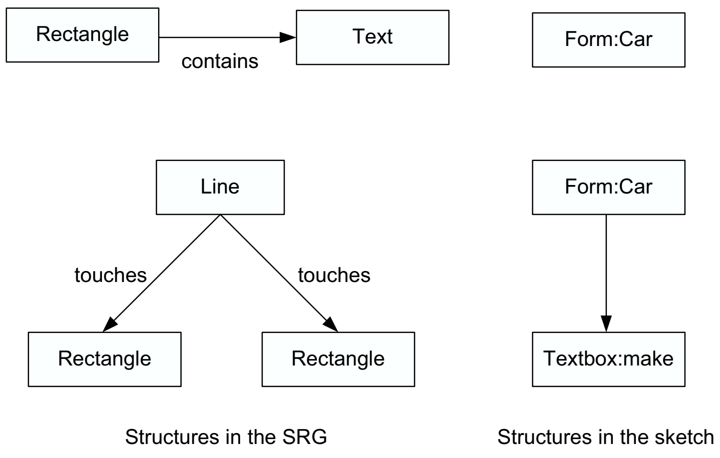


Figure 5.7: UI model structures at the SRG and sketch level

```
...
node:  {title:'1' label:'rectangle'}
node:  {title:'2' label:'v1:car:form'}
node:  {title:'3' label:'rectangle'}
node:  {title:'4' label:'v2:make:textbox'}
node:  {title:'5' label:'line'}
edge:  {sourcename:'1' targetname:'2' label:'contains'}
edge:  {sourcename:'3' targetname:'4' label:'contains'}
edge:  {sourcename:'5' targetname:'1' label:'touches'}
edge:  {sourcename:'5' targetname:'3' label:'touches'}
...
```

```
...
srgnode(1,rectangle).
srgnode(2,[v1,car,form]).
srgnode(3,rectangle).
srgnode(4,[v2,make,textbox]).
srgedge(1,2,contains).
srgedge(3,4,contains).
srgedge(5,1,touches).
srgedge(5,3,touches).
...
```

Having obtained a Prolog representation of the SRG for a UI model sketch, it is possible to parse the SRG to obtain a UI model that can be reasoned about using Prolog. Central to the parsing are the two rules listed below. The `uinode(ID,Name,Type,Y)` rule identifies UI model nodes by looking for retangles that contain text strings. Relationships in the UI model are identified using the `uiedge(Source,Target)` rule. This rule looks for two UI model nodes whose rectangles are touched by the same line. The rule also ensures that the two nodes are not the same (using X \== Y), and that the direction of the relationship is valid using the `legal(Type1,Type2)` rule. To extract a UI model, the rules are applied exhaustively to the prolog representation of the SRG.

```
uinode(ID,Name,Type,Y) :-  srgnode(X,rectangle),;
                           srgedge(X,Y,contains),
                           srgnode(Y,[ID,Name,Type]).

uiedge(Source,Target) :-  uinode(Source,_,Type1,I1),
                          uinode(Target,_,Type2,I2),
                          srgedge(X,I1,contains),
                          srgedge(Y,I2,contains),
                          srgedge(Common,X,touches),
                          srgedge(Common,Y,touches),
                          legal(Type1,Type2),
                          X \== Y.
```

5.3.2 Extracting the Information Content from a UI Model

At the core of the system were the rules shown in Table 5.2 taken from Rollinson and Roberts (1998). The code below gives an illustration of how the rules were implemented in Prolog. The two rules listed below map multi- and single-column combo boxes to abstract types (rules 4 and 6 in Table 5.2). The predicates `type()` and `label()` return the type and the name of a node in the UI model respectively, whilst the predicate `abstract_subset()` returns a list of abstract types which have been mapped from form and groupbox controls.

```
abstract5(Name) :-  label(Vertex,Name),
                    type(Vertex,combobox),
                    multi_column(Vertex).

abstract6(Name) :-  label(Vertex,Name),
                    type(Vertex,combobox),
                    single_column(Vertex),
                    abs_form_grpbox(Result),
                    member(Name,Result).
```

Use of the prolog predicate `findall` was made to find all nodes in the UI model matching a particualar rule. This resulted in three lists of nodes, one for each of the three GSM constructs. Any duplicates contained in the lists were removed and then the elements of each list were "asserted" as facts of the form `type(name,type)`, where `name` is the name of the GSM node and `type` is either abstract, lexical or grouping.

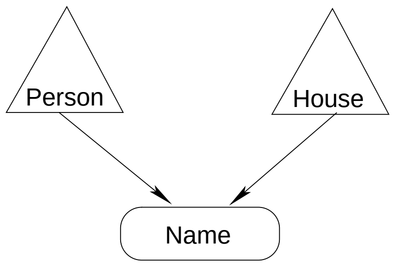
The next step of the process was to extract edges from the UI model and map them to GSM edges. This is achieved as follows: a prolog predicate generates equivalence classes from each of the sets of facts asserted in the previous step. The equivalence classes are asserted into memory in the form `equclass(name,nodes)` where `name` is a UI model label and `nodes` is a set of nodes having `name` as their label. Having asserted the equivalence classes into memory a predicate `gsmEdge(N1,N2)` is defined which takes the labels `N1` and `N2` and tests to see that an edge exists between the

	Abstract Types
1	Forms.
2	Groupboxes containing one or more textboxes, combo boxes, listboxes, grids, groupboxes and checkboxes, but not just checkboxes.
3	Rows containing more than 1 column.
4	Rows containing a single column that have the same name as a form, or a groupbox (as defined in 2).
5	combo boxes with more than 1 column.
6	combo boxes with 1 column that have the same name as a form, or a groupbox (as defined in 2).
7	Textboxes that have the same name as a form, or a groupbox (as defined in 2).
8	A groupbox containing only radio buttons where the groupbox has the same name as a form or a groupbox (as defined in 2).
9	A group containing only checkboxes where the groupbox has the same name as a form or a groupbox (as defined in 2)
	Lexical Types
10	Textboxes and columns.
11	A single checkbox.
12	Groupboxes containing 2 or more radio buttons.
13	Rows with one column that do not have the same name as a form.
14	combo boxes with 1 column that do not have the same name as a form.
	Groupings
15	Groupboxes that contain only checkboxes.
16	Listboxes and Grids.

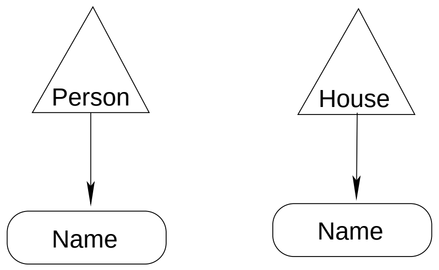
Table 5.2: Informal user interface construct mappings

nodes in the UI model having names `N1` and `N2`. This is achieved by first finding the nodes in the equivalence classes with names `N1` and `N2` and searching for contains relationships between nodes in these classes. By calling the `gsmEdge` predicate with `findall` it is possible to obtain a list of all GSM edges that can then be converted into a set using the builtin predicate `listToSet`. The set of edges are then asserted as facts of the of the form `edge(source,target)`, where `source` and `target` are the names of nodes.

The final step of the extraction process is to "normalise" the GSM schema. By



(a) before normalisation



(b) after normalisation

Figure 5.8: Example of the normalisation process

normalise we mean replace single lexical types that have multiple incoming edges with n copies of each type, one for each incoming edge. Each of the copies is then connected to one of the incoming edges of the original lexical type. Figure 5.8 illustrates a portion of a GSM schema before and after normalisation.

5.3.3 Visualisation of the GSM Schema

To enable easy access to the results of the information extraction process we utilise the XVCG tool (Sander 1995) to view a VCG representation of the GSM schema. To allow visualisation the Prolog facts must be converted into VCG format. This is achieved using a simple Perl script that converts the `type()` and `edge()` predicates into a VCG format file. Figure 5.9 shows XVCG visualising the GSM schema extracted from the uimodel in Figure 5.6(a).

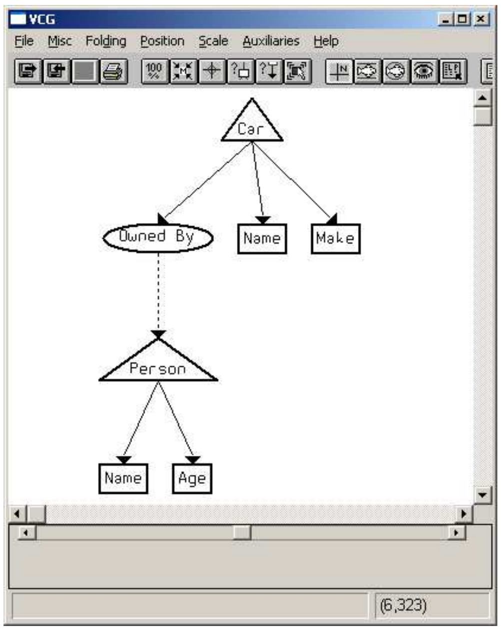


Figure 5.9: Visualisation of the GSM schema using XVCG

5.3.4 Discussion

The implementation of the extraction tool proved useful in that it enabled us to investigate how different data models were able to represent the information conveyed by form interfaces. The tool also represents a major step towards an implementation of a system for comparing the equivalence of form interfaces. As implemented, the system could be used in two ways. The first is for comparing the class of form interfaces that map to the same GSM schema. This could be achieved using the system in its present format. Alternatively, it could be used to compare the class of form interfaces that can be described by the IFO model. Under this scheme it would be possible to compare form interfaces that mapped to the same IFO schema and also those that mapped to distinct IFO schema. The latter could be tested for equivalence by comparing their *normal forms* (see Chapter 7 for a discussion of IFO normal forms).

5.4 Choosing a suitable model

From the previous sections it can be seen that both hierarchic and ER approaches to representing the information conveyed by forms have their benefits. Here, we consider these more closely with the aim of choosing one of the models for our work.

In the introduction to this chapter, we noted that a significant factor in choosing a model would be the existence of research into the problem of equivalence. For all three models such work exists and in this respect, there is little to choose between the three models, however, the NST model is perhaps the weaker of the three as work has only considered the restructuring of this model rather than equivalence directly.

In representing the information conveyed by forms, there are significant differences in the approaches, especially when we consider collections of, rather than, individual forms. For individual forms, each of the three models is suitable with the hierarchic approaches having the advantage due to their close correspondence with the structure of forms. For collections of forms, however, the hierarchic models are limited in comparison with the ER model, especially in situations where the forms are linked in a network-like structure, such as that found in graphical form interfaces. Consider the hypothetical interface comprising three forms shown in Figure 5.10, for example. The arcs represent clickable links between the forms. The student form contains a list of modules. Details of each module can be viewed by clicking on the appropriate row of the listbox which opens the modules form. The modules form contains a textbox holding the lecturers name. Finally, the lecturer form contains a listbox of tutorial students for that lecturer, clicking a row of the listbox opens the student form.

Figures 5.11 and 5.12 show the IFO and the ER representations of the information

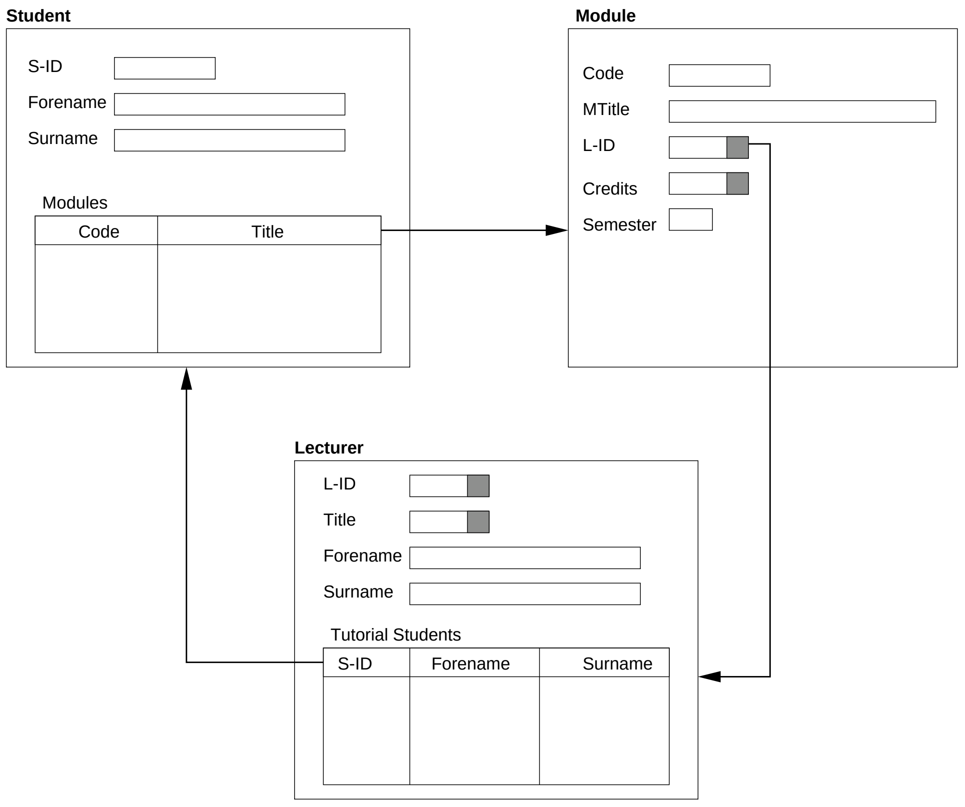


Figure 5.10: A graph structured interface

conveyed by the interface in Figure 5.10. It is clear from Figure 5.11 that the IFO schema is illegal in the sense that it contains a cycle. Thus, the IFO schema is unable to represent the information conveyed by the interface in Figure 5.10. The ER model, however, is able to represent the information conveyed by the interface as is illustrated by Figure 5.12. From this, it would appear that the ER model is the most suitable of the three models for the purpose of representing the information conveyed by graphical form interfaces. The extra flexibility offered by the ER model, however, comes at the expense of a more complicated mapping from the forms interface. For this reason one might argue that the hierarchic approaches should be extended so they are able to deal with the graph structures encountered when modelling collections of forms. Indeed, such generalisations of the these models exist. The logic data model (LDM) (Kuper and Vardi 1993) generalises the IFO model and is capable of representing the information conveyed by the interface in Figure

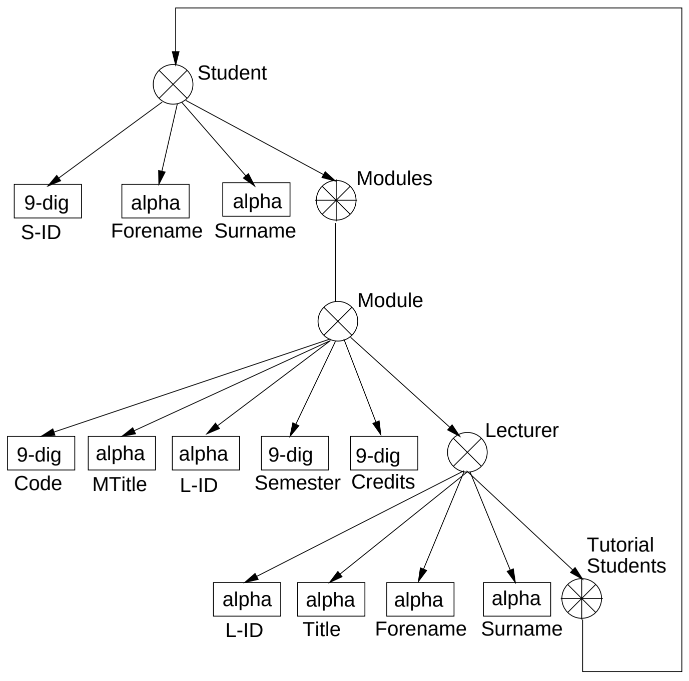


Figure 5.11: IFO schema for Figure 5.10

5.10. In fact the schema in Figure 5.11 is a valid LDM schema. However, whilst the LDM generalises the representational abilities of the IFO model, it does so at the expense of the work on equivalence, which no longer holds in the graph-based logic data model. Thus, the equivalence preserving transformations which would have proved invaluable for comparing the information conveyed by form interfaces are lost.

In light of the discussion above we choose the ER model for the purpose of representing the information conveyed by graphical form interfaces. A number factors influenced this decision: first, the model has been used previously for collections of forms albeit not in a GUI environment; second, it is sufficiently expressive to be able to capture relationships between objects that may occur in a graphical forms interface; and third, a rich body of knowledge exists into comparing the equivalence of ER schemata.

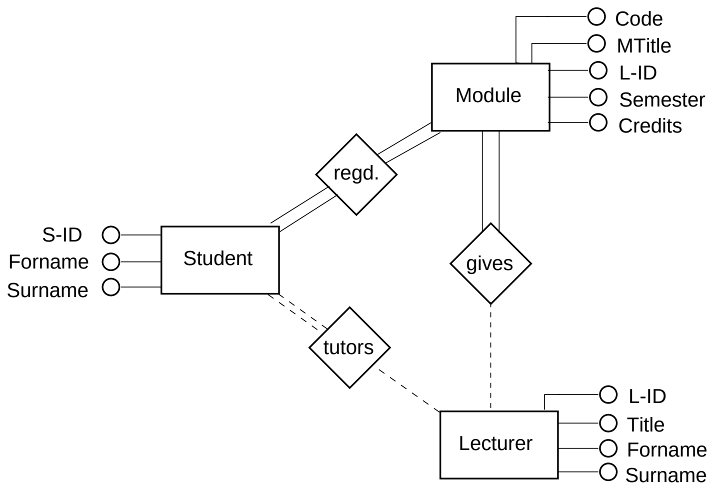


Figure 5.12: ER schema for Figure 5.10

5.5 The Entity-Relationship model for interfaces

The extended Entity-Relationship (EER) model described in this section is our own, however, we borrow the diagramming conventions from the ERC+ model of (Spaccapietra and Parent 1992), which allows both the cardinality and optionality of relationship-types to be representated graphically. Before presenting the definition of our extended ER model, we briefly review the main concepts of the ER approach.

An entity-type is a set of real world objects which share common properties. A STUDENT entity-type, might, for example, have the properties STUDENT-NUMBER and SURNAME. An instance of an entity-type is known as an entity. The entity e_1, whose values for the properties STUDENT-NUMBER and SURNAME are 1220 and “Smithson” repsectively, is an entity of type STUDENT. Entity-types need not be disjoint. The entity-types POSTGRADUATE and UNDERGRADUATE, representing postgraduate and undergraduate students, may both be subsets of STUDENT. Any entity which is of type POSTGRADUATE is also of type STUDENT. An entity-type whose entitities are a subset of another entity-type are called entity subtypes.

A relationship-type is a mathematical function over n entity-types and a relationship is a single instance of a particular relationship-type. Consider, for example, the entity-types STUDENT and DEPT and the relationship type BELONGS = [STUDENT $\times$ DEPT]. If e_1 is an entity of type STUDENT and e_2 is an entity of type DEPARTMENT then the tuple (e_1,e_2) is an instance of the relationship-type BELONGS could be $[e_2, e_1]$. Relationships possess two properties. The first is degree which indicates the number of instances of an entity-type participating in a relationship to the instances of the other entity-types participating in the relationship. If the relationships BELONGS had a degree of many-to-one (written N:1) then this means that several students can work in the same department but a STUDENT can work in only one department at a time.

The second property, optionality, refers to whether an entity-type must participate in an instance of a relationship-type. If, for example, a STUDENT had to be assigned to a DEPARTMENT then the entity-type STUDENT would be described as having *mandatory* participation in the relationship. If, however, students did not have to be assigned to a department then this would be described as *optional* participation.

Domains are either sets of single values, or sets of sets. The values "Computer Science", "Chemistry" and "Physics", might, for example, be classified as belonging to the domain DEPT-NAMES, whilst the following might represent part of the domain MODULE-CODES: { {db11, so12}, {is12, tc11, fp14}}. The former are called single value domains and the latter set-valued domains

Attributes can be one of two types. Atomic attributes are functions mapping from an entity-type into a single valued domain. Set-valued attributes are multi-valued functions mapping from an entity-type into a set valued domain. The property EMP-NO, for example, is an example of an atomic attribute. A set-valued attribute of STUDENT might be MODULES which maps to the domain MODULE-CODES.

It is also possible in this model for attributes to be mandatory or optional. A mandatory attribute means that a value of null does not appear in the attribute's domain of values, whilst an optional attribute does have a null value in its domain.

5.5.1 ER Schemas

We are now in a position to present the ER model formally in preparation for Chapter 6 in which the mapping between the forms interface model and the ER model will be defined.

We assume the sets $\mathcal{E}$ and $\mathcal{R} = \mathcal{E} \times \mathcal{E}$ to be sets of entity-types and binary relationship-types respectively. We assume the set $\mathcal{D}$ to be the set of domains made up of the disjoint union $\mathcal{D}_{single} \uplus \mathcal{D}_{set}$ of single and set valued domains. If d is a domain in $\mathcal{D}$ then $dom(d)$ is the set of values of d excluding the value *null*. If d is suffixed with a $+$, written d^+ then $dom(d^+) = dom(d) \uplus \{\text{NULL}\}$.

We also assume the existence of the set $\mathcal{A} = \mathcal{A}_{atomic} \cup \mathcal{A}_{set}$ which is the set of attributes formed by the disjoint union of atomic and set-valued attributes. Each attribute a_i in $\mathcal{A}_{atomic}$ is a function of the form $a_i : \mathcal{E} \rightarrow \mathcal{D}_{single}$, and each attribute a_j in $\mathcal{A}_{set}$ is a function of the form $a_i : \mathcal{E} \rightarrow \mathcal{D}_{set}$.

Definition 5.1 (Entity-Relationship Schema) *An Extended ER schema is a tuple $S = \langle E, R, A, card, part \rangle$ where:*

- $E \subseteq \mathcal{E}$ *is the set of entity-types;*
- $R = \langle E \times E \rangle$ *is the set of relationship-types between entity-types;*
- $A = A_{\text{atomic}} \cup A_{\text{set}} \subseteq \mathcal{A}$ *is the set of attributes;*

- *card* : $R \rightarrow \{1:1, \mathrm{N}:1, 1:\mathrm{N}, \mathrm{M}:0, \mathrm{M}:\mathrm{N}\}$ *is a function mapping from relationship-type to relationship degree; and*

- *part* : $R \rightarrow \{\langle \mathrm{m}, \mathrm{o}\rangle, \langle \mathrm{o}, \mathrm{m}\rangle, \langle \mathrm{m}, \mathrm{m}\rangle, \langle \mathrm{o}, \mathrm{o}\rangle\}$ *is a function mapping from relationship-type to participation constraints.*

□

The simple ER schema below models a university with departments comprising undergradate and postgradute students. Undergraduates are registered for several modules and postgraduate students may or may not have a thesis.

$$\begin{aligned}
\langle E = \{ \quad & \mathit{student}, \mathit{department}, \mathit{undergrad}, \mathit{postgrad}\}, \\
R = \{ \quad & \langle \text{ student, dept}\rangle\} \\
A_{\text{atomic}} = \{ \quad & \mathit{surname} : \text{student} \rightarrow \text{surnames}, \mathit{student\text{–}no}\text{: } \mathit{student} \rightarrow \mathit{student\text{-}numbers}, \\
& \mathit{thesis} : \text{postgrad} \rightarrow \text{theses}^{+}, \\
& \mathit{name} : \text{department} \rightarrow \text{deptnames}\} \\
A_{\text{set}} = \{ \quad & \mathit{modules} : \text{undergrad} \rightarrow \text{modules}\} \\
\mathit{card} = \{ \quad & \langle \text{ student,dept } \rangle \rightarrow \text{n} : 1\}\rangle \\
\mathit{part} = \{ \quad & \langle \text{ student,dept } \rangle \rightarrow \langle \mathrm{m}, \mathrm{o}\rangle\}\rangle
\end{aligned}$$

Figure 5.13 shows the graphical representation of the ER schema. Subtypes are indicated using directional arcs toward the supertype.

5.5.2 EER Diagrams

The EER schemas from the previous section can be represented pictorially using EER diagrams. Figure 5.13 illustrates the main constructs in the EER diagram. Entity-types and relationship-types are represented using the familiar rectangle and

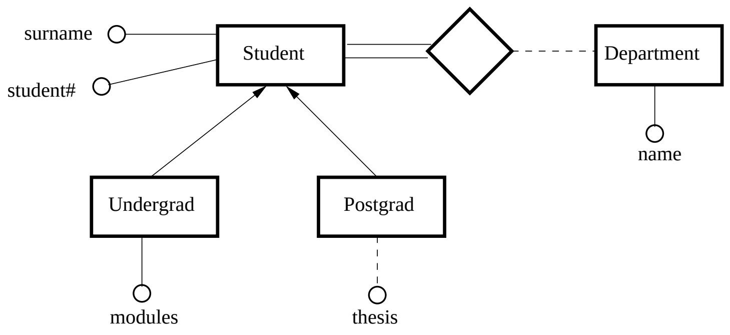


Figure 5.13: A simple ER schema

diamond notations whilst attributes are represented using circles. Cardinality and participation constraints are represented using a notation borrowed from the ERC+ model (Spaccapietra and Parent 1992). A cardinality of many is represented using a double line and a cardinality of one with a single line. Solid lines indicate mandatory participation and dashed lines optional particiaption. This notation is also extended to attributes: double lines representing multi-valued attributes and dashed/solid lines representing attributes that may or may not accept the value NULL.

Formally, an EER diagram can be represented as a graph with nodes corresponding to entity-types and attributes and edges to relationships between entity-types and attributes. We define the notion of an EER diagram in preparation for Chapter 6 that will map from a forms interface schema to an EER diagram.

Definition 5.2 (EER Diagram) *Let an EER diagram be a graph* $ERD = \langle N, E \rangle$ *where:*

- $N = N_E \cup N_A$ *is the set of nodes made up of the set of entity-types* N_E*; and the set of attributes* N_A*. Each attribute* a_i *in* N_A *has an associated domain* d_i*;*
- $E = E_R \cup E_S \cup E_A$ *is the set of edges:*
  - $E_R = \langle N_E \times \{\mathit{1{:}1,\ N{:}1,\ 1{:}N,\ M{:}0,\ M{:}N}\} \times \{\langle m,o \rangle, \langle o,m \rangle, \langle m,m \rangle, \langle o,o \rangle\} \times$

$N_E\rangle$ *is the set of edges representing relationship-types between entity-types.*

- $E_S = \langle N_E \times 1 : 1 \times o \times N_E\rangle$ *is the set of edges representing subtype relations.*
- $E_A = \langle N_E \times \{1, N\} \times \{o,m\} \times N_A\rangle$ *is the set of edges representing attribute relations. If* $e_a = \langle e, c, p, a\rangle\rangle$ *is an edge in* E_A *and* d *is the domain of* a *then* $p = o$ *if* ***NULL*** $\in dom(d)$ *otherwise* $p = m$*; and if* $d \in \mathcal{D}_{set}$ *then* $c = N$ *otherwise* $c = 1$.

- $\lambda : N \rightarrow W$ *is a function that assigns an entity-type/attribute a name from the set of object names*

$\square$

5.6 Summary

This Chapter has investigated three data models for the purpose of representing the information conveyed by a form interface: the IFO model; the Nested Sequence of Tuples (NST) model; and the Entity-Relationship model. These models were chosen, in particular, because restructuring mechanisms or equivalence metrics were available for them and they had all previously been used for the purpose of representing the information conveyed by forms. The availability of equivalence metrics would be beneficial when comparing the information content of forms, whilst the history of being used in some kind of form analysis suggested they may also be useful in the analysis of graphical form interfaces.

To aid in the investigation of how the information conveyed by forms could be represented in a data model a software tool was developed. The tool was able to take a graph-like description of a form interface, and extract from this a Generalized Semantic Model (GSM) schema that represented the information conveyed by the form interface. The use of a general data model like the GSM allowed us to map onto the three data models identified for investigation and gain an appreciation of each was able to represent the information conveyed by graphical form interfaces.

It was found that the network-like structure of graphical form interfaces made hierarchic data models, such as NST and IFO, unsuitable for the purpose of representing the information conveyed by these interfaces. As a result, the ER model was chosen and a definition of our Entity-Relationship model along with several extensions was given.

Chapter 6

Modelling the Information Content of Forms

6.1 Introduction

This chapter aims to capture the process of *recovering* the information conveyed by a graphical form interface in a manner that could be implemented in a computer system. The result of the recovery process will be an extended Entity-Relationship (EER) schema that represents the information conveyed by the interface. The graph representations of forms and information are exploited to employ a graph-rewriting approach to the recovery process. The recovery process transforms a form interface schema into an EER schema using a collection of graph-based productions.

The chapter begins by introducing the material necessary to understand the graph-rewriting approach used and also constitutes a brief review of the graph-rewriting literature. The recovery process is then described in detail and contrasted with the approach of pure graph rewriting. Finally, consideration is given to issues of efficiency with a discussion of how application specific features can be exploited to improve efficiency.

6.2 Graph Rewriting Systems

6.2.1 Introduction

Graph rewriting allows complex manipulations of graphs to be performed through the matching and replacement of subgraphs within the graph being manipulated. Since its conception in the late 1960s graph rewriting has been widely studied, motivated by application areas such as pattern recognition, compiler description, and the specification of databases and abstract data types (Andries, Engles, Habel, Hoffman, Kreowski, Kuske, Schürr, and Taentzer 1996). Indeed, there are numerous examples of graph rewriting used in practice. In the database field, for example, GOOD (Gyssens, Paredaens, den Bussche, and Van Gucht 1994) and Spider (Rodgers and King 1997) both employ graph rewrite rules to specify manipulation of data stored in graph-oriented database management systems. At the University of Leiden, graph rewriting has been used extensively in the IPSEN project (Engels, Lewerentz, Nagl, Schäfer, and Schürr 1992) to develop integrated software development environments. A result of the IPSEN project was the PROGRES graph rewriting language (Schürr, Winter, and Zündorf 1999). PROGRES is capable of being used for a number of applications. Graph rewriting has also been employed in document image analysis for recognition of circuit diagrams (Bunke 1982), mathematical formulae (Grbavec and Blostein 1995) and music scores (Fahmy and Blostein 1993), among others.

At a general level a graph rewriting system operates on a class of graphs using a set of *rewrite rules.* The graph that is manipulated by the rewrite rules is known as the *host graph* and the manner is which the rules are applied is governed by a particular *rule organisation.* In the following sections each of these three components will be examined in detail.

6.2.1.1 Host graph

The hostgraph upon which a particular set of rewrite rules operate will belong to a class of graphs which all share the same properties. The graphs may be: either directed or undirected; cyclic or acyclic; have labels on their nodes and edges; and possibly have attributes assigned to their nodes and edges. IFO schemas (Abiteboul and Hull 1987), for example, represent a class of directed, acyclic graphs, with labelled nodes and edges, and node attributes.

6.2.2 Graph Rewrite Rules

A graph rewrite rule replaces one subgraph in a host graph with another subgraph. To do this, the rule needs to specify: the subgraph to be replaced; how the subgraph will be replaced; and what the subgraph will be replaced with. The subgraph to be replaced is represented as the left-hand side (LHS) of a rule whilst the subgraph that will replace it is represented as the right-hand side (RHS) of the rule. How the rule will be replaced is captured using an *embedding specification.* A rewrite rule

$R_i = (G_{\text{Left}}, G_{\text{Right}})$ is applied to a hostgraph G_{Host} as follows: A match is sought for a subgraph G'_{Left} of G_{Host} that is isomorphic to G_{Left}. If such a subgraph is found, then it is removed, resulting in a graph G_{Rest}. Exactly how the graph is removed depends upon the embedding specification. A graph G'_{Right} that is isomorphic to G_{Right} is then added to G_{Rest}. How the connections are made between G_{Rest} and G'_{Right} is again dictated by the embedding specification. In addition to the test for isomorphism the host graph may have to meet other conditions. A hostgraph with attributes on its nodes may have to satisfy constraints on these attributes, for example. Extra constraints such as these are known as *application conditions* and will be discussed in Section 6.2.4.2

The embedding specification associated with a rule performs the task of connecting the graph G'_{Right} to the host graph. The embedding specification does this by taking the edges that connected the graph, G'_{Left}, to the host graph and converting them to the edges connecting the graph, G'_{Right} to the host graph. For a complete review of embedding mechanisms see (Nagl 1979).

A example of a generalised rewriting step is illustrated in the following example. Consider the graph shown in Figure 6.1 which shall be the host graph and the production in Figure 6.2 that is to be applied to the host graph. The first step is to remove the LHS of the rule from the host. This results in the rest graph shown in Figure 6.3. Notice that the dangling edge is retained in the rest graph. The fact that the edge is retained is specified in the embedding description. For the production in Figure 6.2 the embedding specification is captured graphically by representing edges that should be retained with dashed arcs. In the LHS of the production the dashed arc indicates the edges that should be retained, whilst in the RHS the dashed arc indicates the edge that should be used to “embed” or attach the RHS to the rest graph. The graph resulting from the rewrite operation is shown in Figure 6.4.

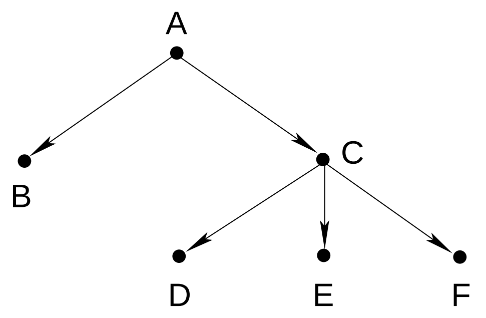


Figure 6.1: Host graph

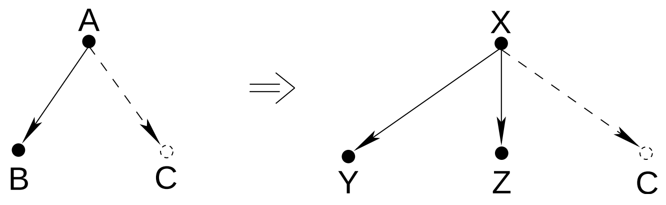


Figure 6.2: A production

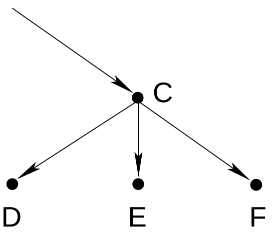


Figure 6.3: The rest graph after removing the LHS of Figure 6.2 from the graph in Figure 6.1

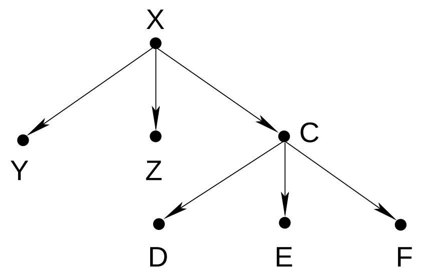


Figure 6.4: The result of applying Figure 6.2 to the graph in Figure 6.1

6.2.3 Rule Organisation

A set of rewrite rules may be organised using one of several strategies. The particular strategy used dictates the type of rewriting system. Rewriting systems which do not order the set of rules are known as an *unordered* rewriting systems; those which order the rules are called *ordered* rewriting systems; and those systems which apply rewrite rules in response to particular actions are known as *event driven* rewriting systems. The *graph grammar* is a special form of rewriting system. Graph grammars differ from other approaches to rewriting in that the rules can be used to generate a language of terminal graphs (see section 6.2.3.4) whereas a graph rewriting system transforms an instance of a given class of graphs into another instance of the same class of graphs (Schürr, Winter, and Zündorf 1995).

The choice of ordering mechanism affects the number of rewrite rule applications that must be tried during the rewriting process and, therefore, has a significant impact on the efficiency of a rewriting system Given that the problem of finding one graph within another is NP-complete it is clear that to be efficient the ordering mechanism should aim to keep the number of rule applications to a minimum.

6.2.3.1 Unordered Rewriting Systems

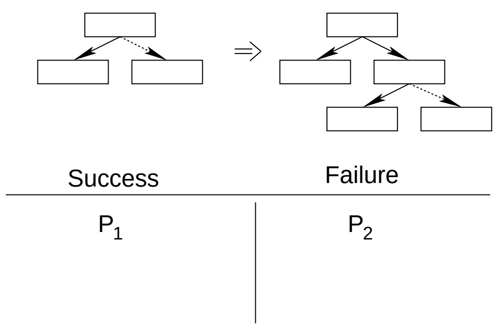
Unordered rewriting systems represent the simplest mechanism for graph rewriting. Rules are chosen at random from the set of rules and applied to the host graph. Depending on the problem, rewriting may terminate when no more rule applications can be made, alternatively rewriting may contine indefinitely. Given that the problem of finding a subgraph isomorphism is NP complete, the "brute force" approach of unordered rewriting means that it is inefficient, especially where there are a large number of rules, because of the number of tests that need to be carried out.

6.2.3.2 Ordered Rewriting Systems

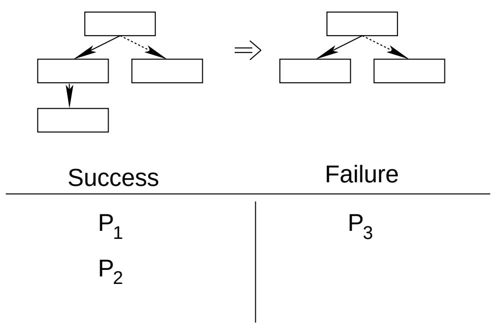
Ordered rewriting systems may be either completely or partially ordered. In a completely ordered system, rules are ordered in a similar fashion to the statements in a programming language. Partially ordered systems may order groups of rules and apply rules within each group in an unordered fashion. The control specification, which dictates the ordering of rules is central to an ordered rewriting system and may be specified in a textual or diagrammatic form. The control specification also dictates the circumstances under which rewriting should terminate.

Whilst completely ordered rewriting mechanisms can be more efficient than unordered rewriting, due to their ability to transform an input graph directly into an output graph, they do this at the expense of simplicity as control specifications can be difficult to formulate and understand. Partially ordered rewriting systems offer a compromise because within groups, the rules remain unordered. Efficiency is also improved by grouping rules as the number of rules that may be applicable at any one time is reduced.

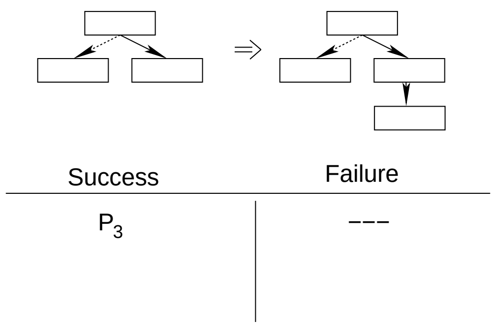
The ordering of rewrite rules has been described using both textual and diagrammatic forms. Textual specifications associate success and failure lists with each production. The success list maintains a list of productions to try if the current production is successfully matched in the host graph and the failure list productions to try if the current production is not matched in the host (see for example (Fu 1982)). Figure 6.5 shows three example productions and their respective success/failure lists. An alternative, diagrammatic, representation of rule ordering is shown in Figure 6.6. The ordering described in Figure 6.6 is the same as that of the success and failure lists shown in Figure 6.5. The diagram is interpreted as follows. Apply rules P_1 and P_2 until they can be applied no more and then apply rule P_3 until no more applications can be made.



(a) Production P_1



(b) Production P_2



(c) Production P_3

Figure 6.5: Productions ordered using success and failure lists

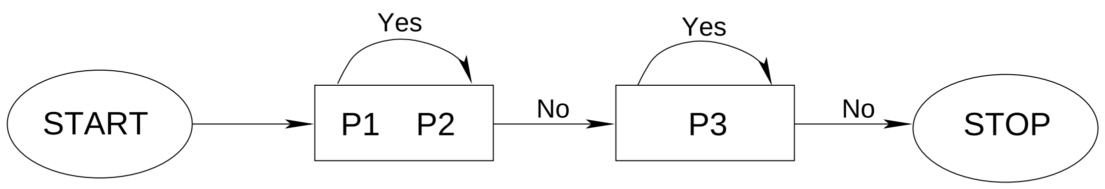


Figure 6.6: A daigrammatic control specification

6.2.3.3 Event-driven Rewriting Systems

A system employing event driven rewriting does not specify on ordering over the set rules. Rather rules are invoked according to the occurance of particular events. Event driven rewriting has been popular in diagram editors and graph-oriented database management systems. In diagram editors, for example, rewriting operations are invoked to update a diagram's underlying graph representation when the diagram is altered. In graph-oriented DBMSs, the addition of records invokes rewrite rules to update the database's instance graph.

6.2.3.4 Graph Grammars

Unlike the rewriting approaches discussed above, the graph grammar approach requires the symbols that are used to label graphs to be classified as either terminal or non-terminal symbols. Grammars can be used for either generation or recognition (parsing). A grammar used for generation transforms a start graph into a terminal graph (i.e. a graph in which all labels are terminal symbols). The set of terminal graphs that can be generated from the start graph, using the rewrite rules, are the language of the grammar. A grammar used for parsing can proceed in a top-down or bottom-up fashion. Top-down parsing attempts to obtain a target graph (the graph to be parsed) by applying rewrite rules to the start graph. Bottom-up parsing applies the rewrite rules to the target graph in an attempt to obtain the start graph.

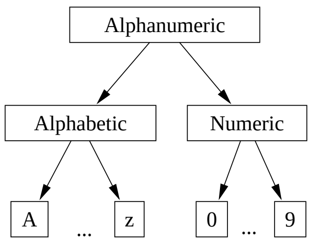


Figure 6.7: A label hierarchy

6.2.4 Extensions to Rewriting Systems

Blostein, Fahmy, and Grbavec (1995) identified several extensions to basic graph rewriting systems: hierarchic organisation of labels; application conditions; node and edge label variables; parameterized rewrite rules; and variable structures

Use of these extensions allows much richer specifications to be captured in rewrite rules and can in many cases reduce the number of rules required. Extensions, however, increase the complexity of the rewrite rules and can also adversely affect the efficiency of the rewriting system.

6.2.4.1 Hierarchic organisation of labels

Normally the symbols used to label nodes and edges are specified as sets and, when comparing two graphs for isomorphism, the labels as well as the structure must match. More flexibility can be obtained by structuring variables hierarchically in classes and subclasses. This results in a tree structuring if inheritance among classes is restricted to single-inheritance, or lattice structures where multiple inheritance is permitted.

Figure 6.7 illustrates an example label hierarchy for character and integer labels. When matching graphs using hierarchic labels, rather than requiring labels to match in the two graphs a match occurs if a label in one graph occurs at or below the level of the label in the other graph.

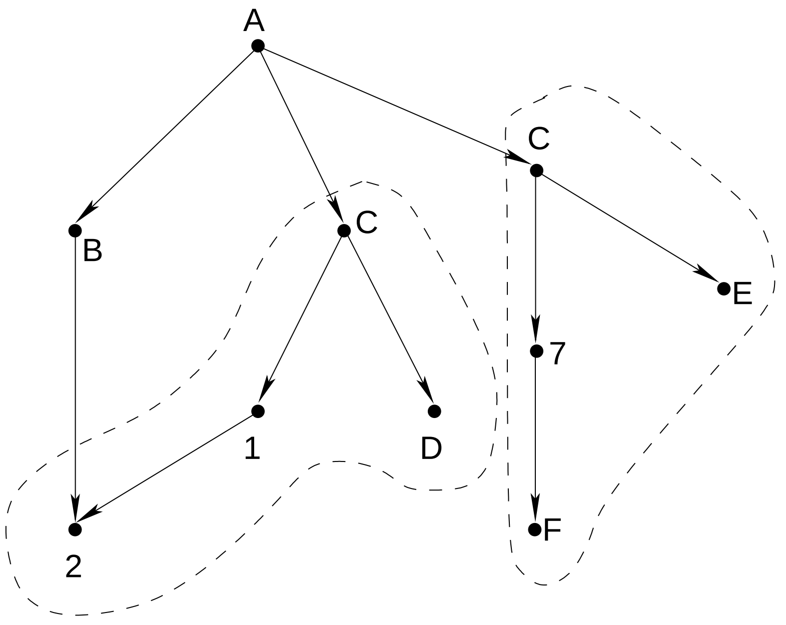


Figure 6.8: Subgraphs matched under heirarchic labelling

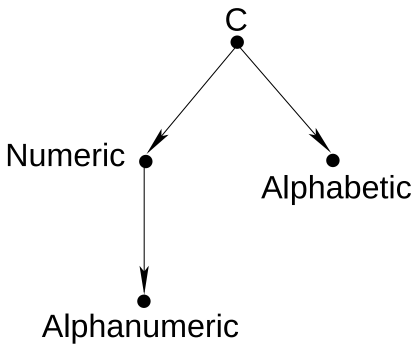


Figure 6.9: The LHS of a production using hierarchic labels

Consider the host graph shown in Figure 6.8 and the production shown in Figure 6.9. Under the label hierarchy shown in Figure 6.7 the production would match to the two subgraphs shown in Figure 6.8 by the dashed lines. The two subgraphs match because the nodes labelled “I” and “D” and “7”, “E” appear below the labels “Numeric” and “Alphabetic” in the label hierarchy and the nodes labelled “2” and “F” appear under the label “Alphanumeric”. In effect, matching under a label hierarchy is akin to having sets of labels in the LHS of a production and performing a membership rather than equality test when comparing node labels.

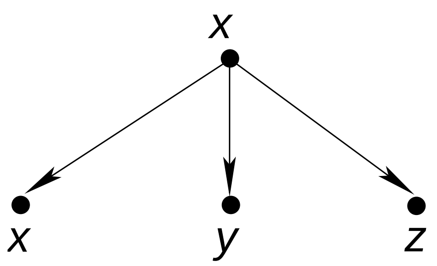
6.2.4.2 Application conditions

When a rewrite rule is applied to a host graph the isomorphism test may be insufficient, especially where node or edge attributes need to be examined. Application conditions, therefore, provide a means by which other constraints such as attribute values can be tested. Depending on the constraint, application conditions can be specified textually, diagrammatically or as a combination of both. In Δ-notation, Loyall and Kaplan (1992), for example, diagrammatic specification is used to represent the graphical constraint which dictates that parts of the LHS graph that must be absent from the host for a match to occur. Textual specification is used in PROGRES (Schürr, Winter, and Zündorf 1999) as a means of placing constraints on attribute values.

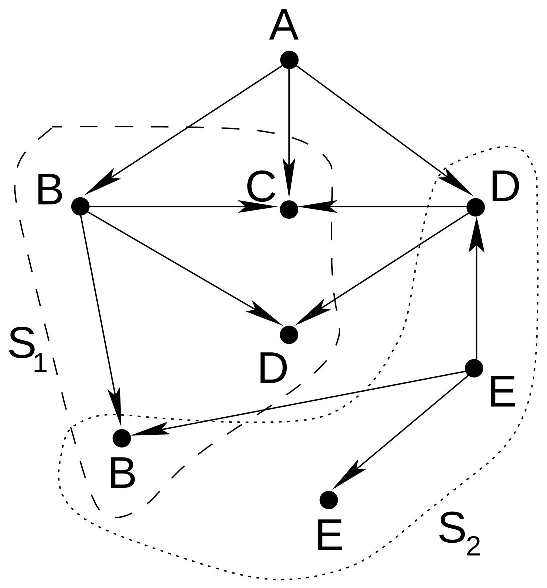
6.2.4.3 Variable labels and structure

Variable node and edge labels allow the number of rewrite rules to be reduced significantly because a single rule can be applied in a number of different situations. Rather than labelling primitives in the pre- and post-condition graphs with constant symbols variables are used as labels. Constants can be bound to variables either before or during the isomorphism test.

When using variable for node labels it is possible to assign values to the variables before performing the subgraph isomorphism or during the isomorphism. The example in Figure 6.10 shows how a single production (note that only the LHS of the production is shown) whose nodes are labelled using variables can be used to perform the same task a multiple productions whose nodes are labelled using literal values. The production, when applied with the two sets of bindings $S_1 = \{x/\mathrm{B}, y/\mathrm{D}, z/\mathrm{C}\}$ and $S_2 = \{x/\mathrm{E}, y/\mathrm{B}, z/\mathrm{D}\}$, can be used in place of the two rules in Figure 6.11.

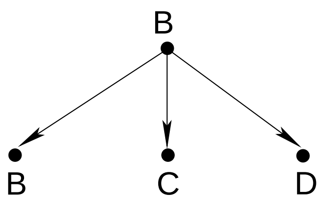


(a) LHS of a production with variables for node labels

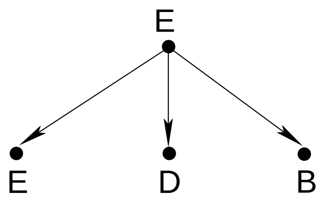


(b) Matching subgraphs

Figure 6.10: Subgraph matching when using variables for node labels



(a)

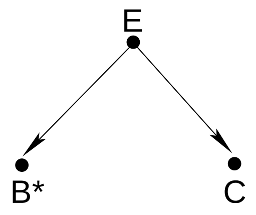


(b)

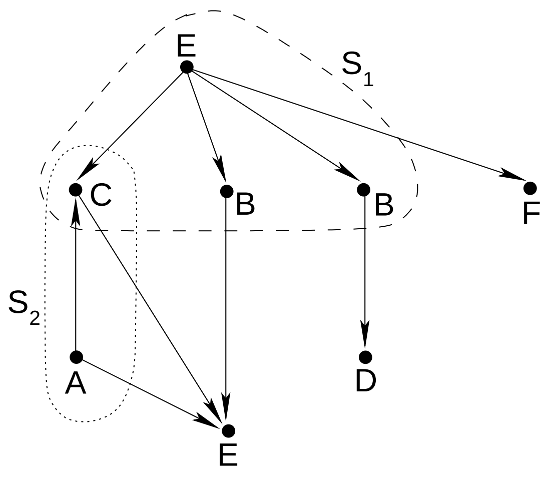
Figure 6.11: LHS of productions that do the same as that in Figure 6.10(a)

Some rewriting approaches, most notably Δ-notation, allow the graphs in the rewrite rule to contain repeating nodes and edges. Allowing variable structures can, like variable labels, reduce the number of rewrite rules required in a rewriting system. Figure 6.12 illustrates the use of productions that allow matches with varying structure. The asterisk by the node in the production shown in Figure 6.12(a) indicates that the node in the LHS of the production can match zero, one or more nodes in the host graph. When applied to the host graph in Figure 6.12(b) the production matches the subgraphs outlined by the dashed lines. The match with subgraph S_1 illustrates the production matching with one or more nodes in the host, whilst the match with subgraph S_2 illustrates matching where there are no occurrances of the repeating node in the host graph.

The use of variables in rewriting systems can increase the size of the search space considerably. Rewriting systems that make use of variables, therefore, must be done in a controlled manner if the system is to run efficiently.



(a) LHS of a production with wildcard nodes



(b) Matching subgraphs

Figure 6.12: Matching subgraphs of varying structure

6.2.4.4 Parameterized rewrite rules

Paramterized rules are commonly used in ordered or event-driven rewriting systems. This is because rules are invoked by certain events or control instructions, and the use of parameters allows information associated with an event, for example, to be passed to a rewrite rule. Parameterized rewrite rules are often used in event driven rewriting, especially where variable node labels are used that can be specified as parameters to the rule. Where a rewrite rule uses parameters to assign values to node variables then the rewrite rules operate in the same manner as the example shown in Figure 6.10. Alternatively, if no paramaeters are specified then the variables are bound during the isomorphism test thus allowing the LHS of a production to adapt itself to any host graph environment.

6.2.5 Graph rewriting as a means of computation

Despite the age of the field of graph rewriting, there still remains a number of issues which need to be resolved if graph rewriting is to be used in practice. Foremost of these is that of efficiency. Zündorf (1993) identifies two main efficiency problems: (i) the testing of rules for applicablity; and (ii) the subgraph-isomorphism test.

Rewriting systems employing a large number of rules can be inefficient because at each rewriting step all the rules need to be tested for applicability. One solution, adopted in the PROGRES language, is to reduce the number of applicable rules by specifying which rule(s) may be applied after the current rule has executed (Zündorf 1993). In PROGRES this, in some cases, results in only a single rule being applicable and thus affords a significant increase in efficiency.

Subgraph-isomorphism testing is, in general, an NP-complete problem and has receieved a significant amount of attention from the graph rewriting community. Dörr (1995) and Bunke *et al.* (1991) among others have proposed efficient algorithms for subgraph-isomorphism testing. In addition to these general approaches to improve the efficiency of isomorphism testing, it is also possible to exploit properties peculiar to the application area. An application that employs graphs with labelled nodes and edges can, for example, reduce the search space for isomorphism testing considerably (Blostein, Fahmy, and Grbavec 1995).

Graph rewriting provides a powerful mechanism for manipulating graphs and allows operations to be performed by specifying the operations themselves as graphs thus retaining the high level representation abilities of graphs for both problem space and computation. If designed carefully graph rewriting systems need not suffer unecessarily from efficiency problems. Indeed, much work is being undertaken into improving the efficiency of graph rewriting (see for example Bunke *et al.* (1991) and Schürr, Winter, and Zündorf (1999)).

6.3 Capturing Information Content

Representing the form interface and information content as graphs enables the use of techniques from graph rewriting to extract the information content from a form interface schema. The approach taken, however, is not one of pure graph rewriting in which the input graph and the output graph belong to the same class of graphs. The approach presented in this chapter uses two distinct classes of graph. The input graphs belong to the class of form interface schemata (FIS), described in Chapter 4, whilst the output graphs are the class of extended entity-relationship schemata described in Chapter 5. Matching, therefore, takes place in the form interface schema whilst modification occurs in the extended entity-relationship schema.

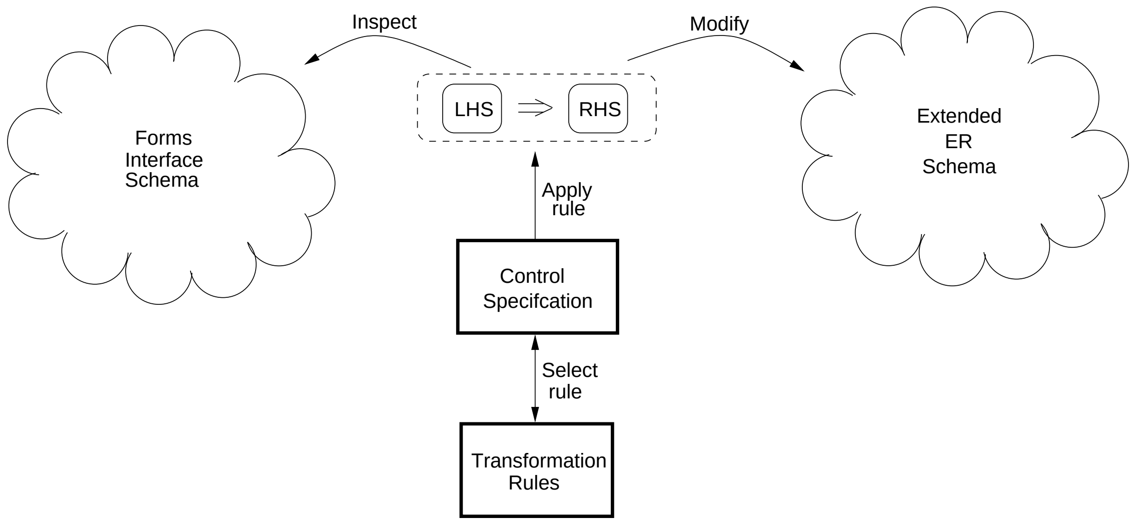


Figure 6.13: Overview of the recovery process

Figure 6.13 gives an overview of the mapping process. Rules are applied to the FIS in a particular order as dictated by the control specification. The control specification also ensures that the rules are applied exhaustively and that the transformation process terminates.

In the following sections, we examine the rules and control specification comparing and contrasting them to those used in a pure graph rewriting approach. We start by considering the rules.

6.4 Rules

There are 76 rules in total (see appendix A) and each rule is classified according to the structure that it tranforms:

- *entity-type* rules extract entity-types and identifying attributes (primary keys) from the form interface;
- *atomic attribute* rules extract single-valued attributes of entity-types from the

form interface;

- *multi-valued attribute* rules are used to extract set-valued attributes from the form interface;
- *relationship* rules are divided into four sub-groups that extract 1:1, 1:N, N:1 and M:N relationships from the form interface schema;
- *complex structure* rules extract E-R structures involving several entity- and relationship-types from the form interface; and
- *subtype* rules identify entity subtypes from the form interface

6.4.1 Structure

Each rule comprises a left-hand side (LHS) and a right-hand side (RHS). The LHS is a collection of rooted trees specified using the primitives of the form interface model whist the RHS is an EER schema. Formally a rule is represented as a pair $R = \langle \text{LHS}, \text{RHS} \rangle$.

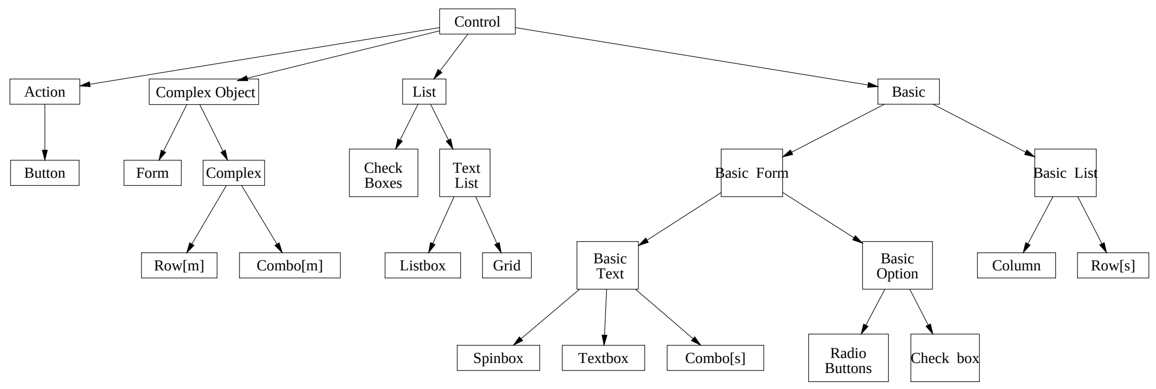
6.4.1.1 Left-hand side

The left-hand side of the rule specifies the structure that is being matched and any context structures required by the match. The former is known as the focus structure and the latter the context structure. Unlike the LHS of a pure graph rewriting rule, the LHS of rules in this method comprise several rooted trees. The tree representing the focus structure is called the focus tree and the tree(s) that represent the context structure are known as context trees. The set of context trees is further divided into *required* and *prohibited* sub-contexts. The required subcontext contains trees that must be present in the host graph for a match to take place whilst the prohibited subcontext contains trees that must be absent from the host graph for a match to take place.

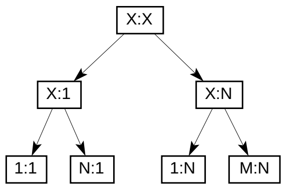
Three extensions are required to the basic primitives of the form interface model to enable rules to specified in a succinct and general fashion. These were briefly introduced in Chapter 4. The first extension extends the naming scheme of nodes to include variables. The name of a node, therefore, can be either a symbol representing a variable (we use uppercase letters from the end of the alphabet for this purpose) or a constant value. The reason for employing variables is that it would be impossible to write rules that dealt with every possible naming scheme for every rule. Structures must, therefore, be matched independently of their name, however, names are important to the matching process because they are used to identify the occurance of certain structures in an FIS. By using variables to name the nodes in rules, it is possible to create rules that are both general in that they can be applied in regardless of node names; and that, through the naming of several nodes with the same variable, capture the naming schemes which signify the presence of particlar information structures.

Variables are also used to transfer names from the LHS to the RHS. The RHS will always have a set of variables that are a subset of or equal to the set of variables in the LHS. Upon finding a match, all the variables in the LHS will have been bound to a constant values. These bindings are used to name EER structures.

In the mapping between a form interface schema and an EER schema it is often the case that several similar structures in the FIS map to the same structure in the EER schema. They may, for example, differ only in the type of particular controls. Situations also arise in which the cardinality of the edge does not affect the mapping or where the edge condition can be relaxed to x:1 X:N, where x is either 1 or M.



(a) Control types

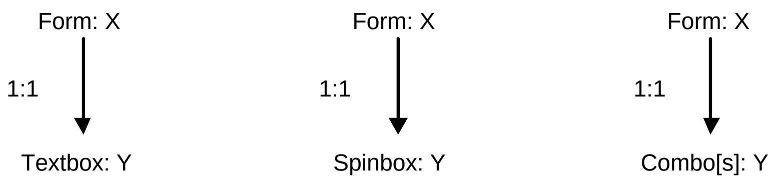


(b) Cardinalities

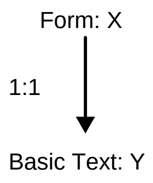
Figure 6.14: Label hierarchies

Some form of *wildcard* mechanism is, therefore, required. The second extension to the form interface model, hierarchic labelling, fulfills these requirements. Figure 6.14(a) shows the hierarchy of control types and Figure 6.14(b) the hierarchy of edge cardinalities. A further advantage of using hierarchic labelling is that the number of rules is reduced because a single rule can do the job of several. Consider, for example, the three rules shown in Figure 6.15(a). Using the rule in Figure 6.15(b) the three transformations can be described using a single rule.

Prohibited sub-context trees provide a means of specifying structures that should be absent from an FIS for a particular rule to fire. These, however, work at the level of the structure rather than at the individual control level, and in addition to being able to prevent matching if complete structures exist, we also require this type of functionality at the control level. The third extension, therefore, introduces the notion of negative and positive graph primitives (Rodgers and King 1997) to the



(a)



(b)

Figure 6.15: Reducing the number of rules with hierarchic labels

form interface model. Under this scheme, which we call *polarity*, nodes and edges are classified as being either positive or negative. Postives nodes and edges must exist during matching whilst negative ones must be absent.

Formally, we define the notion of a *control tree* to represent the trees that appear in the LHS of a rule.

Definition 6.1 (Control Tree) *A control tree, T, is a form interface schema (see page 62), such that:*

- *T is a rooted tree;*
- *If N is the set of nodes in the control tree and $n \in N$ is the root node then $\tau(n) \in \{\text{Form}, \text{Tuple}\}$; and*
- *the nodes of T are labelled using the members of Σ_{V}, the set of symbols representing variables.*

$\square$

Using Defintion 6.1 we can define the LHS of a rule. Formally, the LHS of a rule comprises a control tree designated as the *focus tree* and a set of control trees representing the global context of the transformation. As mentioned earlier the *context* is divided into two sub-contexts. The *required* sub-context comprises control trees that should be present in the FIS during matching whilst the *prohibited* sub-context represents structures that should be absent.

Definition 6.2 (Left-hand side of a rule) *The left-hand side of a rule* LHS $= \langle T_{\mathrm{F}}, \mathcal{T}_{\mathrm{C}} \rangle$ *is pair where* T_{F} *is the focus tree and* $\mathcal{T}_{\mathrm{C}} = \mathcal{T}_{\mathrm{C}+} \cup \mathcal{T}_{\mathrm{C}-}$ *is the set of context trees such that:*

- T_{F} *is a control tree;*
- $\forall T_i \in \mathcal{T}_{\mathrm{C}}$ *then* T_i *is a control tree*

□

6.4.1.2 Right-hand side

The right-hand side of a rule comprises an EER schema. The only difference between the EER schema in the rule and that representing information content is the use of variables to label nodes and edges in the rule.

Definition 6.3 (Rule) *The right-hand side of a rule* RHS = ERD $= \langle N, E, \lambda^{\mathrm{RH}} \rangle$ *is an extended Entity-Relationship diagram where* $\lambda^{\mathrm{RH}} : N \rightarrow \Sigma_V$ *is a function labelling nodes with variables.*

□

6.4.2 Rule Application

The application of a rule involves two main steps: inspection and modification. To apply a rule, the left-hand side of the rule is taken and a match is sought in the host graph. If a match is found then the right-hand side is used to replace the LHS in the host graph. To enable a mapping between a form interface and EER schema the basic rule application procedure requires some extension. Firstly, to facilitate the match of focus and context trees and secondly to ensure the transfer of values from the LHS to the RHS of the rule.

Several extensions are actually needed to allow inspection (matching) to take place, these are: the notion of a tree match that incorporates both edges that should be present and those that should be absent; application conditions to test bound and unbound variables used on names; an application condition to test the constraints on controls; and the notion of a context match which incorporates *required* and *prohibited* sub-contexts.

We now discuss how the extensions have been included in the matching process to form the *inspection* stage of rule application.

6.4.2.1 Inspection

The inspection process is based around the matching of trees to subgraphs in the form interface schema. Matching must deal with trees that contain:

(i) only positive primitives; and

(ii) both negative and positive primitives.

The first case requires an ismorphism to be found between the nodes and edges in the tree and the nodes and edges in some sub-schema of the form interface schema which respect labelling, naming and data entry constraints.

The second case is more complex requiring that an isomorphism exist between the positive primitives of the tree and a sub-schema of the FIS; and that there is no isomorphism between a sub-schema of the FIS and either the whole tree, or some subtree that includes at least one negative primitive.

Matching of patterns containing negative primitives follows the approach of Rodgers and King (1997) but which restricts their concept of a graph sub-match to a tree sub-match. A tree sub-match is an isomorphism between a subset of the nodes and edges in a control tree and a subschema of the FIS. If a tree sub-match is performed on only the positive primitives in a control tree, and the control tree only contains positive primitives then provided an isomorphic subschema can be found in the FIS then a tree sub-match occurs. A rule that contains both negative and positive primitives matches if and only if there is a tree sub-match of the positive primitives; and there is no subschema in the FIS isomorphic to the whole control tree; and no subtree matches that include at least one negative node. Depending on the number of negative nodes in the control tree there may have to be several isomorphism tests between the different combinations of these.

In addition to dealing with nodes that should be absent from a form interface schema the rule must also ensure that any matching subschema respects the naming constraints of the rule. To ensure this each pair of nodes in the rule with the same variable name must have an isomorphic pair of nodes in the FIS with the same constant names. Furthermore, each pair of nodes with distinct variable names must have distinct constant names.

Before defining a tree submatch formally several functions are required to deal with hierarchic labelling and the various application conditions.

If e is an edge in the set of edges, E, of a form interface schema/control tree then the following functions are defined: $\mathit{type}(e) : E \rightarrow \{\text{opens}, \text{contains}\}$ returns the type of the edge; $\mathit{source}(e) : E \rightarrow N$ returns the source of the edge; and $\mathit{dest}(e) : E \rightarrow N$ returns the destination of the edge. For an edge, e, the function $\mathit{card}(e) : E \rightarrow C$ returns the cardinality of the edge

If the sets of control types T and T_{w} are structured in a hierarchy, as shown in Figure 6.14(a), and $\mathit{level}(t)$ is the depth of a type from the root of the hierarchy, then two types, t_1 and t_2, are equal (written $t_1 \equiv t_2$) if $\mathit{level}(t_1) \leq \mathit{level}(t_2)$ and there exists a path, $p_t = (t_j, \ldots, t_k)$, such that $t_1 = t_j$ and $t_2 = t_k$. Similarly, if the set of edge cardinalities, C, is structured in a hierarchy as shown in Figure 6.14(b), then two cardinalities, c_1 and c_2, are equal (written $c_1 \equiv c_2$) if $\mathit{level}(c_1) \leq \mathit{level}(c_2)$ and there exists a path $p_c = (c_j, \ldots, c_k)$, such that $c_1 = c_j$ and $c_2 = c_k$.

Let T be a tree in the LHS of a rule, V be the set of variables occurring in T and Σ_{O} the set of object names. Let $B = \{v/w \mid v \in V \ \wedge \ w \in \Sigma_{\text{O}}\}$ be a set of bindings for the variables in T and let $\beta(v)$ be a function returning the object name bound to the variable v.

Definition 6.4 (Tree submatch) *Given a form interface schema* $\text{FIS} = \langle N^{\text{FIS}}, E^{\text{FIS}}, \lambda^{\text{FIS}}, \tau^{\text{FIS}}, \psi^{\text{FIS}} \rangle$*; a control tree* $T = \langle N^{\text{T}}, E^{\text{T}}, \lambda^{\text{T}}, \tau^{\text{T}}, \psi^{\text{T}} \rangle$*. Let* N^{T}_{C} *be a subset of* N^{T}*; and* E^{T}_{C} *be a subset of* E^{T}*.*

A tree submatch is a pair of injective functions $(\pi_{\text{N}}, \pi_{\text{E}})$ *where* $\pi_{\text{N}} : N^{\text{T}}_{\text{C}} \rightarrow N^{\text{FIS}}$ *and* $\pi_{\text{E}} : E^{\text{T}}_{\text{C}} \rightarrow E^{\text{FIS}}$ *such that:*

- $\forall n \in N^{\text{T}}_{\text{C}}, \psi^{\text{T}}(n) \subseteq \psi^{\text{FIS}}(\pi_{\text{N}}(n)) \ \wedge \ \tau^{\text{T}}(n) \equiv \tau^{\text{FIS}}(\pi_{\text{N}}(n))$*;*

- $\forall e \in E^{\mathrm{T}}_{\subset}, source(e) = source(\pi_{\mathrm{E}}(e)) \ \wedge \ dest(e) = dest(\pi_{\mathrm{E}}(e))$*;*
- $\forall e \in E^{\mathrm{T}}_{\subset}, card(e) \equiv card(\pi_{\mathrm{E}}(e))$*;*
- $\forall e \in E^{\mathrm{T}}_{\subset}, type(e) = type(\pi_{\mathrm{E}}(e))$*; and*
- $\forall n_1, n_2 \in N^{\mathrm{T}}_{\subset}, \lambda^{\mathrm{T}}(n_1) = \lambda^{\mathrm{T}}(n_2) \Leftrightarrow \lambda^{\mathrm{FIS}}(\pi_{\mathrm{N}}(n_1)) = \lambda^{\mathrm{FIS}}(\pi_{\mathrm{N}}(n_2))$.

$\square$

Using the concept of a tree submatch it is possible to define a tree match that accommodates both single (i.e positive) and mixed polarity trees.

Before defining a tree match formally, the concepts relating to trees containing negative primitives must be defined. In particular sets of trees containing the different combinations of negative nodes. Each tree in this set must contain at least one negative node, and each negative node must be directly connected to a positive node via an incoming edge. If $T = \langle N^{\mathrm{T}}, E^{\mathrm{T}} \rangle$ is a tree containing negative nodes then the set $neg(T)$ is the set of trees where each tree $T' = \langle N^{\mathrm{T}'}, E^{\mathrm{T}'} \rangle$ is a tree such that $N^{\mathrm{T}}_{+} = N^{\mathrm{T}'}_{+}$, $E^{\mathrm{T}}_{+} = E^{\mathrm{T}'}_{+}$, $N^{\mathrm{T}}_{-} \subset N^{\mathrm{T}'}_{-}$ and $E^{\mathrm{T}}_{-} \subset E^{\mathrm{T}'}_{-}$. If T_1 is a tree in $neg(T)$ then there is no other tree T_2 in $neg(T)$ such that $T_1 = T_2$. A tree $T_1 = \langle N, E \rangle \in neg(T)$ is said to be minimal iff $\forall e = (t, c, y, h) \in E-, t \in N+$. The set of trees $neg(T)$ is miminal iff $\forall T \in neg(T), T$ is minimal.

Definition 6.5 (Tree match) *Let FIS be a form interface schema, T be a control tree and $neg(T)$ a minimal set of negative trees for T. A tree match, written $match(T, FIS)$, is defined as a tree submatch (τ_N, τ_E) of $N^{\mathrm{T}}_{+}, N^{\mathrm{FIS}}, E^{\mathrm{T}}_{+}$ and E^{FIS} where for any tree $T_{-} = (N^{\mathrm{T}}_{-}, E^{\mathrm{T}}_{-}) \in neg(T)$ there is no tree submatch (ϕ_N, ϕ_E) of $N^{\mathrm{T}}_{-}, N^{\mathrm{FIS}}, E^{\mathrm{T}}_{-}$ and E^{FIS} where:*

(i) $\forall n \in N^{\mathrm{T}}_{+}, \phi_N(n) = \tau_N(n)$*; and*

(ii) $\forall e \in E_{-}^{\mathrm{T}}, \phi_E(e) = \tau_E(e)$.

□

Finally, we are able to specify the conditions under which a rule matches using the definitions developed thus far in this section. Recall that the LHS of a rule comprises a focus tree and a set of context trees, divided into two subsets: required and prohibited. For a rule to match successfully with a subschema of an FIS then it must be that:

(i) the focus tree is present in FIS;

(ii) all of the *required* sub-context is present in FIS;

(iii) none of the *prohibited* sub-context is present in FIS; and

(iv) the naming scheme is respected.

A rule match is, therefore, a simple problem of using a tree match to test the focus tree and all of the context trees. Each sub-context is tested separately. The *required* context is tested by iterating over the set of trees and peforming a tree match on each to ensure that a match exists for each tree in the sub-context. Testing the *prohibited* context iterates over the set of trees and upon finding a tree in the FIS causes the rule to fail. The naming scheme of a rule is tested by comparing each binding of a variable in the focus tree with bindings of the same variable (if any) in the context trees. If any of the bindings of the context tree variables do not match the binding of the focus tree variable then the naming scheme is not respected and the rule fails. These processes are defined formally below.

To test the set of context trees the required and prohibited sub-contexts have to be treated differently. A *for all* test is used to test the required context, hence:

$$reqmatch(\mathcal{T}_{\mathrm{C}+}, \mathrm{FIS}) \Leftrightarrow \forall T_{\mathrm{C_i}} \in \mathcal{T}_{\mathrm{C}+}, match(T_{\mathrm{C_i}}, \mathrm{FIS}).$$

For the prohibited sub-context an *exists* match is used to first test for the existence of any of the trees, the result is then negated to ensure that when none of the trees in the prohibited sub-context exist we return true. Hence:

$$\neg cpmatch(\mathcal{T}_{\mathrm{C}-}, \mathrm{FIS}) \Leftrightarrow \exists T_{\mathrm{C_i}} \in \mathcal{T}_{\mathrm{C}-}, match(T_{\mathrm{C_i}}, \mathrm{FIS}).$$

Testing the naming scheme of a rule is achieved by comparing the bindings of the focus tree with those of each context tree. For this purpose we define the functions: *bmatch* which returns true if two sets of variable/constant bindings are equivalent; and *namingscheme* which returns true if for all the context trees in a rule the variable/constant bindings are equivalent to the variable constant bindings of the focus tree. Let:

- $bmatch(\mathcal{B}_1, \mathcal{B}_2)$ return true if $\forall v_1/c_1 \in \mathcal{B}_1 \;\; \forall v_2/c_2 \in \mathcal{B}_2, (c_1 = c_2) \vee \neg(v_1 = v_2)$ where $\mathcal{B}_1, \mathcal{B}_2$ are sets of variable/constant bindings; and
- let $namingscheme(T_{\mathrm{F}}, \mathcal{T}_{\mathrm{C}})$ return true if $\forall \mathcal{T}_{\mathrm{C_i}} \in \mathcal{T}_{\mathrm{C}}, bmatch(\mathcal{B}(\mathcal{T}_{\mathrm{C_i}}), \mathcal{B}(F))$.

Rules occurring in the rule set can be classified as one of three types according to the contents of the set of context trees. The type of a rule dictates the exact procedure that is used to to perform the match:

(i) *local rule* - a local rule comprises a focus tree only and, therefore, requires only the matching of the focus tree;

(ii) *global required rule* - a global required rule has a focus tree and a *required* sub-context. Matching this type of rule requires a tree match of the focus tree along with a tree match for **all** of the context trees;

(iii) *global prohibited rule* - a global prohibited rule has a focus tree and a *prohibited* sub-context. Matching this type of rule requires a tree match of the focus tree and there to be no matches for **any** of the context trees.

A rule match is defined as follows:

Definition 6.6 (Rule match) *Let* FIS *be a form interface schema; R be a rule where* $\mathrm{LHS_R} = \langle T_\mathrm{F}, \mathcal{T}_\mathrm{C}\rangle$ *is the left-hand side of R; and let* $\mathrm{LHS} \sqsubseteq \mathrm{FIS}$ *signify a "match" of the left-hand side of R with the form interface schema* FIS *such that:*

- $\mathrm{LHS} \sqsubseteq \mathrm{FIS} \Leftrightarrow \mathit{type}(R) = \mathrm{local} \ \wedge \ \mathit{match}(T_\mathrm{F}, \mathrm{FIS})$
- $\mathrm{LHS} \sqsubseteq \mathrm{FIS} \Leftrightarrow \mathit{type}(R) = \mathrm{global+} \wedge \mathit{match}(T_\mathrm{F}, \mathrm{FIS}) \wedge \exists \mathit{cmatch}(\mathcal{T}_{\mathrm{C}+}, \mathrm{FIS}) \wedge \mathit{namingscheme}(T_\mathrm{F}, \mathcal{T}_\mathrm{C})$
- $\mathrm{LHS} \sqsubseteq \mathrm{FIS} \Leftrightarrow \mathit{type}(R) = \mathrm{global-} \wedge \mathit{match}(T_\mathrm{F}, \mathrm{FIS}) \wedge \neg \mathit{cpmatch}(\mathcal{T}_{\mathrm{C}-}, \mathrm{FIS}) \wedge \mathit{namingscheme}(T_\mathrm{F}, \mathcal{T}_\mathrm{C})$

□

This concludes the inspection step of a rule. In the next section the modification procedure is described as well as the mechanism by which the values of attributes in the LHS are transferred to the RHS before addition to the EER schema.

6.4.2.2 Addition

The modify operation as shown in Figure 6.13 on page 109 does not alter the FIS host graph rather it modifies the EER schema. This is unlike *pure* graph rewriting in which the occurence of the left-hand side in the host graph is replaced by the right-hand side of the rule. Whilst our scheme removes the need for a complex embedding mechanism that has to detatch and re-attach occurrences of the LHS and RHS of a rule from the host graph, it introduces the problem of detecting when a particular occurrence in the FIS has been matched and ensuring that the strcuture is not matched again. If we do not keep a record of successful matches then an infinite loop may occur as the same occurrence of a LHS may be matched numerous times, thus causing the transformation process to continue indefinitely. One solution would be to remove the matched occurrence from the FIS, however, it may act as context for other matches and its removal may lead to inconsistencies in the transformation process. A better solution is to apply each rule only once and during that application locate all occurrences of the left-hand side of a rule in the FIS. This, combined with the notion a "local" delete, in which the deletion of structures in the FIS is limited to a a particular form for the application of a specific rule, ensures that no occurrence can be matched more than once. Upon completion of a rule application the "locally deleted" structures are replaced in the FIS. Alogrithm 6.1 illustrates the procedure for finding all occurrences of the left-hand side of a rule in a form interface schema.

Algorithm 6.1 (Finding all occurrences of a rule)

```
let $FIS_r$ be a copy of the form interface schema $FIS$
repeat
  find $LHS \sqsubseteq FIS_r$
  if match found then
    $add(RHS, ERD)$
    remove all nodes in $LHS$ from $FIS_r$ except for FORM nodes
  end if
until no match for $r$ can be found
```

The actual addition of the RHS to the EER schema requires a check to be made so that any primitives common to the RHS and EER schema are not added again. The primitives to be added, therefore, are identified by comparing primitives in the RHS with these in the FIS. Entity-types and attributes are identified by checking that their name does not appear in the FIS. Relationship-types are identified by checking that a relationship does not already exist between the same two entity-types in the FIS. Where the relationship does exist it can be brought to the users attention who are asked to resolve the conflict by choosing to ignore the relationship or adding it to the EER schema as an additional relationship.

Formally, we define three sets $Ents, Rels$ and $Attrs$ to hold the entity-types, relationship-types and attributes to be added. The function $\beta(l)$ returns the constant that is bound to the the label l.

If $RHS = \langle N^{\mathrm{RH}}, E^{\mathrm{RH}}, \lambda^{\mathrm{RH}} \rangle$ is the ERD on the right-hand side of the rule and $ERD = \langle N, E, \lambda \rangle$ is the ERD representing the information content of the interface then the sets of entity-types, relationship-types and attributes to add to ERD are given by the following:

- $Ent = \{n : n \in N_{\mathrm{E}}^{\mathrm{RH}} \ \wedge\ \forall n' \in N_{\mathrm{E}},\ \ \beta(\lambda^{\mathrm{RH}}(n)) \neq \lambda(n')\}$;
- $Rel = \{r : r = (e_1, e_2) \in E^{\mathrm{RH}} \ \wedge\ \forall r' = (e_1', e_2') \in E,\ \ \beta(\lambda^{\mathrm{RH}}(e_1)) = \lambda(e_1') \ \wedge\ \beta(\lambda^{\mathrm{RH}}(e_2)) = \lambda(e_2') \ \wedge\ (part(r) \neq part(r') \ \vee\ card(r) \neq card(r'))$; and
- $Attr = \{a : (e, a) \in E^{\mathrm{RH}} \wedge\ \nexists r = (e', a') \in E,\ \ \beta(\lambda^{\mathrm{RH}}(a)) = \lambda(a') \ \wedge\ \beta(\lambda^{\mathrm{RH}}(e)) = \lambda(e')\}$.

Having identified the sets of entity-types, relationship-types and attributes to be added the following functions are defined to add them to the EER schema, replacing the variable names with their bound constants.

The functions $adde(e, ERD), adda(a, ERD)$ and $addr(r, ERD)$ are used to add members of the respective sets to the ERD describing the information content of the form interface:

- $add_e(e, ERD)$ adds a new entity-type e' to ERD such that $\lambda(e') = \beta(\lambda^{\mathrm{RH}}(e))$;
- $add_a(a, ERD)$ adds a new attribute a' to ERD such that $\lambda(a') = \beta(\lambda^{\mathrm{RH}}(a))$; and
- $add_r(r = (e_1, e_2), ERD)$ adds a new relationship-type r' to ERD where $r' = (e'_1, e'_2)$ such that $\lambda(e'_1) = \beta(\lambda^{\mathrm{RH}}(e_1))$ and $\lambda(e'_2) = \beta(\lambda^{\mathrm{RH}}(e_2))$ and $part(r') = part(r)$ and $card(r') = card(r)$.

Finally, definition 6.7 below uses the functions above to add all primitives to the EER schema.

Definition 6.7 *Let RHS be the right-hand side of a rule and ERD an entity-relationship diagram describing the information content of the form interface schema. $add(RHS, ERD)$ is a function that adds the right-hand side of a rule RHS to the entity-relationship diagram ERD such that add performs the following operations:*

- $\forall e \in Ent, add_e(e, ERD)$;
- $\forall a \in Attr, add_a(a, ERD)$; *and*
- $\forall r \in Rel, add_r(r, ERD)$.

□

6.4.3 Applying a rule

With definitions of inspection and modification in place, it is now possible to define the rule application process. There are two options for applying a rule to an FIS:

(i) find a single occurance of LHS in the FIS; or

(ii) find all occurances of the LHS in the FIS.

The first method is used by many rewriting mechanisms as it is simpler than the second. This is because with the second option, used in approaches such as GOOD (Gyssens, Paredaens, den Bussche, and Van Gucht 1994) and parallel graph rewriting systems, it is possible that several occurances may overlap and when rewriting takes place the effects of replacing the matched subgraphs will be unpredictable. It is possible, however, for us to adopt the second method because we are not performing parallel rewriting and any modifications which could potentially lead to unpredictable results (i.e. multiple relationships between the same entity-types) are flagged to the user anyway.

Definition 6.8 (Rule application) *Let R be a rule, FIS a form interface schema and ERD the entity-relationship diagram describing the information content of FIS. The application of a rule $R = \langle LHS, RHS \rangle$ written $\alpha(R, FIS, ERD)$ is defined as $\forall LHS \sqsubseteq FIS \Rightarrow add(RHS, ERD)$.*

□

6.5 Control Specification

The control specification controls the application of the rules to ensure that the transformation process is complete. This is achieved in two ways. First, by inspecting every form in the interface and second, by applying every rule to each form. In doing this it is possible to say that a transformation is complete with respect to the set of rules. Without the control specification to direct the application of the rules they would be applied in a random order and possible repeatedly to the same structure. Thus, the transformation may not be complete and may not terminate.

To enable a complete transformation and to ensure that the process will terminate, techniques from ordered and event-driven graph rewriting have been employed. The control specification follows the direct transformation approach of ordered graph rewriting by taking an input graph and producing an output graph following a pre-determined sequence of operations. In doing this, cursor nodes from event-driven rewriting are used to guide the application of rules to certain structures in the FIS.

The control specification performs two main tasks. It:

- ensures that each form is processed; and
- applies the set of rules exhaustively to the currently selected form.

To perform these tasks the control specification uses information about the location of forms and complex controls in the interface. These controls are identified using marked, or cursor, nodes. Two sets of marked nodes are maintained, one for forms and the other for complex controls. Each set is partitioned into two subsets: *matched* which is initially empty, and contains matched forms/complex controls; and *unmatched* which contains unmatched forms or complex controls.

The control specification proceeds by taking a form from the set of unmatched forms; applying the rules exhaustively to the form and any unmatched complex controls on the form; and then places the form and any complex controls into the respective matched subsets, removing them from the unmatched subsets. The next form is then chosen from the unmatched set of forms. By identifying and systematically applying the rules to each form node and recording which forms have been transformed it is possible to ensure that all forms have been transformed and that no form has been inspected twice.

Applying the set of rules to a form consists of taking each group of rules and applying each rule in the current group. Once all the rules in a group have been applied the next group is selected and so on, until all the groups have been applied. Having applied each group of rules to the form, any complex controls on the form then become the target of the rules. Again, each group of rules is applied to the complex controls until there are no complex controls remain in the unmatched subset of marked complex nodes.

To demonstrate how the control specification works consider an form interface comprising three forms F_1, F_2 and F_3. Let us assume that form F_1 contains three complex controls C_1, C_2 and C_3 and that form F_3 contains two complex controls C_4 and C_5. Form F_2 has no complex controls on it. The contents of the two sets of marked nodes and their respective subsets are shown below.

$$\begin{aligned} FORMS &= F_{matched} \bigcup F_{unmatched} \\ F_{matched} &= \emptyset \\ F_{unmatched} &= \{F_1, F_2, F_3\} \end{aligned}$$

$$\begin{aligned} COMPLEX &= C_{matched} \bigcup C_{unmatched} \\ C_{matched} &= \emptyset \\ C_{unmatched} &= \{C_1, C_2, C_3, C_4, C_5\} \end{aligned}$$

The control specification begins by taking the first form, F_1, from $FORMS_{unmatched}$ and applying the rules, group by group, to the form. Once all the rules have been applied to F_1, the first complex control on the form, C_1, is taken and all the rules applied to it, followed by the other two complex controls C_2 and C_3. Having applied the rules to form F_1 and all its complex controls the nodes are moved into the relevant matched subsets. The state of the matched and unmatched subsets are shown below.

$$\begin{aligned} F_{matched} &= \{F_1\} \\ F_{unmatched} &= \{F_2, F_3\} \end{aligned}$$

$$\begin{aligned} C_{matched} &= \{C_1, C_2, C_3\} \\ C_{unmatched} &= \{C_4, C_5\} \end{aligned}$$

Form F_2 would then be selected, the rules applied to it, and then moved to the matched subset. The remaining form, F_3, would be selected next and the rules applied to it. Each of the two complex controls on the form would then be selected and the rules applied to them. Finally, form F_3 and the complex controls, C_4 and C_5, would be moved to the matched subsets and the process is then complete. The final state of the sets are shown below.

$$\begin{aligned} F_{matched} &= \{F_1, F_2, F_3\} \\ F_{unmatched} &= \oslash \end{aligned}$$

$$\begin{aligned} C_{matched} &= \{C_1, C_2, C_3, C_4, C_5\} \\ C_{unmatched} &= \oslash \end{aligned}$$

The application of the rules is driven by the way in which they are specified. The focus tree of each root has at its root node a control that is either a form or a complex control. By identifying these structures in the FIS before the transformation process begins, it is possible to direct the search for matches as well as guaranting that all rules have been applied to all forms. Furthermore, the search space for isomorphism testing is much reduced as the focus tree search can be restricted to the current form node and a partial isomorphism exists between the current form node and the root node of the focus tree. Context trees are also rooted at either a form or a complex control. Thus marked nodes can also be exploited to reduce the search space when searching for context trees in the FIS. Algorithm 6.2 illustrates the control specification in pseudocode. The function $des(t, F)$ returns true if the tuple node t is a descendent of the form node F.

6.6 Efficiency Considerations

Efficiency is a central issue in graph rewriting and it is essential if any graph rewriting system is to be of pratical use that the rewriting mechanism be efficient. The main bottleneck in any rewriting algorithm is the time taken to find occurances of the left-hand side of a rule in the host graph (i.e. the isomorphism test). The time complexity for finding all occurances of a single rule, with a left-hand side of size L, in a host graph of size G is G^L (Bunke, Glauser, and Tran 1991).

Algorithm 6.2 (Transformation of a form interface to an EER schema)

Input: A form interface schema
Output: An extended ER schema

```
let $FORMS$ = set of marked nodes of type = Form
let $TUPLES$ = set of marked nodes of type = Tuple
let $RULESET$ = the set of transformation rules

for each form node $F \in FORMS$ do
  for each group of rules $R$ in $RULESET$ do
    for each rule $r \in R$ do
      $\alpha(r, FIS, ERD)$
    end for each rule
  end for each ruleset

  let $TUPLES_{\mathrm{F}} = \{t : t \in TUPLES \wedge des(t, F)\}$
  for each marked nodes $T \in TUPLES_{\mathrm{F}}$ do
    for each group of rules $R$ in $RULESET$ do
      for each rule $r \in R$ do
        $\alpha(r, FIS, ERD)$
      end for each rule
    end for each ruleset
  end for each tuple
end for each form
```

The rules used in this work require several isomorphism tests per rule due to the number of focus and context trees. The time efficiency of a rule is, therefore, $T \cdot G^L$ where T is the number of trees on the left-hand side of a rule. It is possible to exploit the marking of nodes to improve the efficiency of the isomorphism test. The marked nodes can be used as starting points for the isomorphism tests due to the correspondence between the type of the marked node (i.e. form or tuple) and the type of the root nodes of focus and context trees in the rules which are also of the type form or tuple. Thus the partial match removes the need to search the host graph "blindly" for an initial match.

To improve the efficiency of the transformation process, we focus on the properties of the form interface schema and the rules. Before illustrating how the control specification might be altered to yield a more efficient transformation, some basic metrics for comparing efficiency need to be defined. Given that the isomorphism test is central to the efficiency of any graph rewriting system, we choose the number of isomorphism tests required to transform an interface as our efficiency metric.

Given a rule $R = \langle LHS, RHS \rangle$ where $LHS = \langle T_{\mathrm{F}}, \mathcal{T}_{\mathrm{C}} \rangle$ the number of isomorphism tests required to apply the rule is determined by the function $icount(R) = 1 + |\mathcal{T}_{\mathrm{C}}|$. That is, the test required for the focus tree and the several tests required for the context trees. The number of tests, I_r, required for an exhaustive application of all 76 rules is given by equation 6.1.

$$I_r = \sum_{i=1}^{n=76} icount(r_i) \tag{6.1}$$

The exact number of isomorphism tests required to transform a form interface depends on the number of forms in the interface and the number of complex controls each form contains. Recall that during transformation every rule is applied to each form and complex control in the interface to ensure completeness. Equation 6.2

gives the total number of isomorphism tests for a particualar form interface schema f.

$$I_f = (I_r \times n) + \sum_{i=1}^{n}(m_i \times I_r) \tag{6.2}$$

In general, therefore, to transform an FIS with n forms and m_i complex controls on each form requires I_f isomorphism tests. It should be noted that this is the worst case. In practice the number of isomorphism tests required may be smaller due to failed focus tree matches removing the need for testing a rule's context trees.

6.6.1 Partitioning Heuristics

To improve the efficiency of the transformation process we must reduce the number of isomorphism tests required to exhaustively test all of the rules (I_r in Equation 6.2). To achieve this we restrict the rule set to only those rules that are applicable to the current interface structure being transformed. This can be done by either: selecting only rules where the control at the root of the focus tree is of the same type as the structure being transformed (i.e. form or tuple); or examining the depth of a form and using only rules whose focus tree has a depth that is equal to or less than that of the form.

Partitioning rules according to the type of the control at the root of the focus tree is possible because of the way in which form and complex nodes are transformed separately. When processing a form, it is unnecessary to attempt matching rules whose focus tree is rooted at a tuple because a match will never occur. Similarly when processing complex controls, attempting to match rules with a form as the root of the focus tree would also fail on every occassion. Partitioning, therefore, eliminates unnecessary comparisons. Indeed, the number of comparisons can be reduced from 134 to 16 for complex contols.

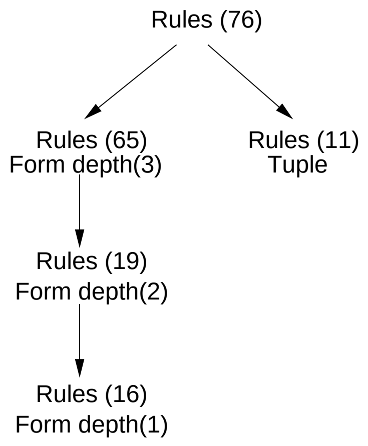


Figure 6.16: Rules partitioned into classes according to focus tree depth

The second type of restriction exploits the nesting of controls which gives forms a *depth*. Forms and focus trees whose root control is a form have a depth determined by the number of controls that are nested on the form. This includes other forms accessed via controls. A form containing a listbox which in turn contains a row which has several columns is said to have a depth of 3 whilst a form containing a button that links to another form is said to be of depth 2. By classifying form rules according to their depth and examining the depth of a form prior to transformation, it is possible to eliminate rules whose depth is greater than that of the form being rewritten. Figure 6.16 shows the classes of form rule depth and the partitioning of rules according to root node type. The figures beside each class show the number of rules in each class. It is clear from Figure 6.16 that significant savings are made in the number of isomorphism tests for forms whose depth is shallower than three. This saving can be illustrated by way of a simple example. Consider a form containing

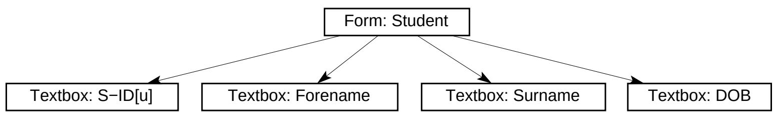


Figure 6.17: Simple interface requiring 135 tests

only textbox controls such as that shown in Figure 6.17. Without partitioning of the ruleset all 76 rules have to be applied to ensure that the transformation is complete. With partitioning, however, only 16 rules need to be applied because we are able to restrict the rule set to those rules with a depth of one.

Algorithm 6.3 (Transformation algorithm including partitioning heuristic)

Input: A form interface schema
Output: An extended ER schema

```
let $FORMS$ = set of marked nodes of type = Form
let $TUPLES$ = set of marked nodes of type = Tuple
let $\mathcal{R}, \mathcal{R}_{\text{tuple}}, \mathcal{R}_{\text{form}}, \mathcal{R}_{\text{form3}}, \mathcal{R}_{\text{form2}}$ and $\mathcal{R}_{\text{form1}}$ be the sets of rules

for each form node $F \in FORMS$ do
  if $depth(F) = 1$ then
    let $RULESET = \mathcal{R}_{\text{form1}}$
  else if $depth(F) = 2$ then
    let $RULESET = \mathcal{R}_{\text{form2}}$
  else if $depth(F) = 3$ then
    let $RULESET = \mathcal{R}_{\text{form3}}$
  else
    let $RULESET = \mathcal{R}_{\text{form}}$
  end if
  for each group of rules $R$ in $RULESET$ do
    for each rule $r \in R$ do
      $\alpha(r, FIS, ERD)$
    end for each rule
  end for each ruleset

  let $RULESET = \mathcal{R}_{\text{tuple}}$
  let $TUPLES_{\text{N}} = \{t : t \in TUPLES \wedge des(t, F)\}$
  for each marked nodes $T \in TUPLES_{\text{N}}$ do
    for each group of rules $R$ in $RULESET$ do
      for each rule $r \in R$ do
        $\alpha(r, FIS, ERD)$
      end for each rule
    end for each ruleset
  end for each marked node
end for each form
```

Using the ideas discussed above a more efficient algorithm (Algorithm 6.3) for performing the transformation process can be devised. The function *depth*(t) in Algorithm 6.3 returns the depth of a tree in the FIS.

6.6.2 Naming Heuristics

In addition to the heuristics that limit the size of the ruleset, another heuristic can be used to prevent the unnecessary testing of context trees after a focus tree has been successfully matched. This heuristic is based on the naming scheme used in rules. It compares the number of times a variable occurs in the context trees, with the frequency of the constant in the FIS to which the variable's occurance in the focus tree is bound. Consider, for example, a form interface schema in which the constant *student* appears only once. The focus tree has matched a subschema in the FIS and the constant student has been bound to a variable X in the focus tree. The variable X also occurs in one of the context trees. This suggests that for the rule to match, the FIS should have two controls with the name student. In the FIS, however, there is only one occurrence of student. It is obvious that a match for the context tree containing the variable X would never be found and performing the test is unnecessary. This heuristic would be implemented at the rule application level. More specifically in Definition 4.6 which would be modified so that comparison of bound variable and constant frequencies took place before the testing of the context trees in the naming scheme.

6.7 Summary

The knowledge required to describe the information conveyed by a graphical form interface in terms of an extended Entity-Relationship schema can be captured as a collection of production rules that map from user interface controls to Entity-Relationship primitives. By employing a graph-based representation of forms and EER schemata we are able to exploit the power of graph rewriting to express the mapping from interface to EER schema. In contrast to textual specifications, the use of graphs and graph rewriting allows the spatial relations, that are central to the mapping, to be expressed easily. Our approach differs from pure graph rewriting in that we operate on a pair of, rather than a single, host graph. The inspection step of rewriting is carried in one of the graphs (the form interface schema) and the modification step in the other graph (the EER schema); because the FIS is never modified, it is necessary to ensure that the same structure in the FIS is never matched more than once, otherwise the mapping process may never terminate. This problem does not affect pure graph rewriting approaches as the matched structure is modified by the during the rule application and thus can never be matched again. To solve the problem we introduce the notion a "local" delete which removes the matched structure for the duration of a rule application replacing it once all occurrences of a rule have been matched in the FIS.

Our choice of graph rewriting as the method by which to express the mapping between interface and EER schemata has meant that we have had to pay careful attention to issues of efficency. Indeed, it is well known that the problem of finding an occurrence of one graph within another graph (the subgraph isomorphism problem) is NP-complete. Whilst finding an efficient method of performing the subgraph isomorphism problem is beyond the scope of this work, we show how it is possible to exploit properties specific to our application to reduce the number of subgraph isomorphism tests to a minimum. Through the marking of particular nodes in the form interface schema graphs we are able to direct the application of the production rules rather than applying them *blindly* to the graph. Furthermore, we also examine the depth of trees in the left-hand side of a rule to reduce the set of rules that are applicable at any one time.

The ability to capture the information content of a graphical form interface in terms of an extended Entity-Relationship schema provides the extraction component of the forms modification system outlined in Chapter 1. In the next chapter we will go on to examine how the information content of two interfaces might be compared.

Chapter 7

Equivalence of form interfaces

7.1 Introduction

The final component of the forms modification system outlined in Chapter 1 is a mechanism for comparing graphical form interfaces for equivalence. This chapter describes how such a mechanism might be provided. In doing so, the conflicts that may occur between interfaces conveying the same information are investigated and presented in the form of a taxonomy. Previous approaches to the problem of form equivalence are also examined and are shown to be insufficient for comparing graphical form interfaces. The main contribution of the chapter is the framework that enables graphical form interfaces to be compared. The framework distinguishes between those interfaces that can be compared using FIS isomorphisms; and those that are compared via their EER schemata. Two definitions of equivalence are identified, the first captures the fact that two interfaces convey the same information. The second definition of equivalence extends the first with the additional condition that if one interface respects user interface guidelines then, provided that the other interface conveys the same information, it will also respect user interface design guidelines.

7.2 Interschema Relationships and Conflicts

Interschema relationships are those relations, either conceptual or logical, that hold between objects in different database schemata which represent the same real world information (Ramesh and Ram 1995). Identifying interschema relationships is particularly important for the integration of schemata in heterogeneous databases as well as for the inter-operation of multi-database systems, as they allow assertions to be made about how a real world structure in one schema is represented in another schema. In a similar way interschema relationships can be specified between form interfaces. From an equivalence point of view interschema relationships are significant because they enable assertions to be made about different structures in an interface that represent the same real world information.

7.2.1 Database schema conflicts

Discrepancies between database schemata that model the same part of the real world arise for a number of reasons. The designer(s) of the systems may have approached the same problem from different perspectives and have used different terminology to refer to the same concepts; different data models may have been chosen to model the problem; and different database management systems used to implement the design. Where different data models have been used, differences can arise due to the expressiveness of the models used. One model may have a richer set of constructs than the other, for example. Even if the same data model has been used by both designers there is still the possibility of discrepancies. Consider a pair of EER schemata, for example. One schema models undergraduates and postgraduates as subtypes of student, whilst in the other schema the student entity-type has an attribute *student-type*. The more constructs there are in a data model the greater the number of ways there are in which to model the same situation (Batini, Lenzerini, and Navathe 1986, page 334).

Batini *et al.* (1986) define four types of relationship between a pair of schemata. Two objects from different schemata are said to be: *identical*, if they are exactly the same; *equivalent*, if structurally different, but equivalent modelling constructs have been used; *compatible*, if they are neither identical or equivalent but there is nothing to suggest they are different; and *incompatible* if the two are clearly different. The type of interschema relationship has an effect on whether two objects can be seen as representing the same real world concept. When the interschema relationship describes the objects as identical then there is no doubt that they represent the same real world concept, however, if the objects are described as equivalent or compatible then there is a difference in their representation and the objects are said to be in conflict (Spaccapietra and Parent 1998). To aid in the identification of similar structures that are represented differently, taxonomies of conflicts have been developed. Kim, Choi, Gala, and Scheevel (1995) distinguish between schema level conflicts (i.e. entity-type and attribute conflicts) and instance level conflicts (i.e differences in scale and precision). By enumerating possible conflicts, it is possible to identify the areas for which methods of comparison or equivalence metrics need to be defined. In the case of the equivalence interschema relationship, one must define a set of criteria under which two constructs can be said to be equivalent.

This may be as simple as a one-to-one mapping or may require a more complex approach employing a series of atomic transformations that preserve equivalence (Batini *et al.* 1986). Examples of the latter approach can be found in (Hull and Yap 1984) and (Abiteboul and Hull 1988) who propose sets of equivalance preserving transformations for the FORMAT and IFO models respectively.

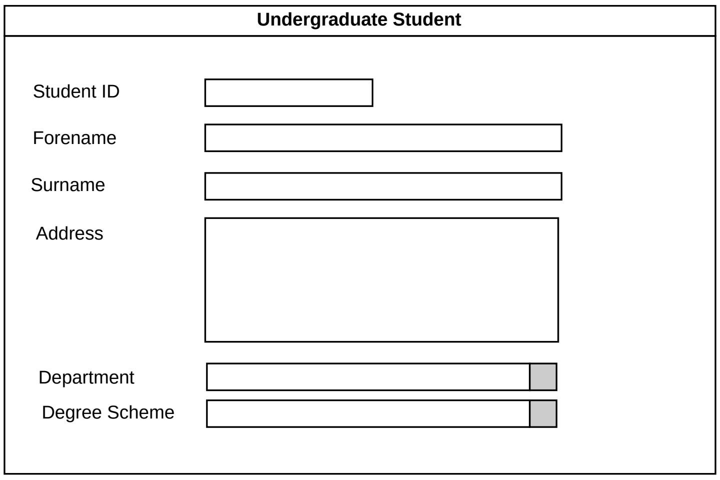
For the purpose of this work, developing a taxonomy of differences between forms will help to identify areas which need to be addressed when developing conflict resolution methods for form interfaces. In the next section, the differences between two forms are highlighted by means of examples and are used to formulate a taxonomy of differences between form interfaces.

7.2.2 Form interface conflicts

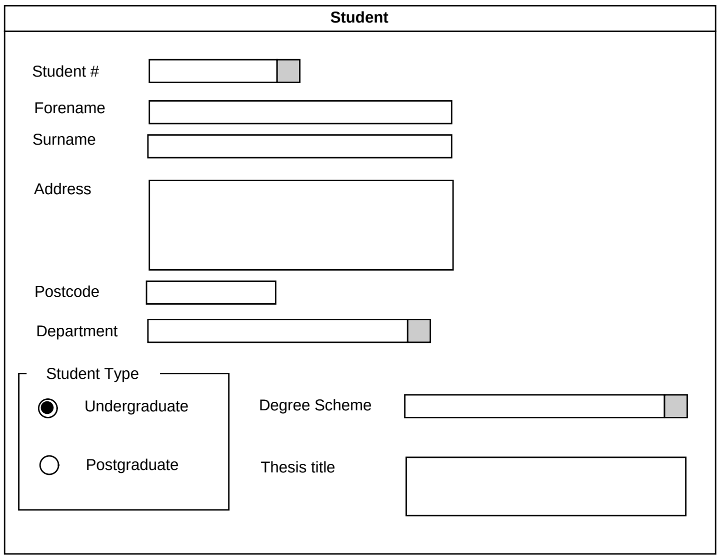
The constructs from which a form interface is built can be classified as either forms or fields. A taxonomy of differences among form interfaces will, therefore, capture differences between forms and fields. Within these two classes differences may exist among individual forms/fields or among groups of forms/fields. The former are known as single form/field conflicts and the latter as multi- form/field conflicts. The discussion that follows examines the two interfaces in Figure 7.1 and illustrates how conflicts may occur.

7.2.2.1 Form Conflicts

Upon comparison of the two forms in Figure 7.1, several differences are immediately apparent: their names differ; the layouts differ; and fields appearing on one form do not appear on the other. These differences are all classed as single form conflicts.



(a) Form 1



(b) Form 2

Figure 7.1: Two forms exhibiting single form differences

In spite of the conflicts that exist between the two forms in Figure 7.1, it is clear that some kind of relationship exists between them. An obvious interpretation of the fields suggests that the relationship is one of subsumption which might be confirmed by examining the two forms at the record level. Assuming that the interface control used to represent a field has no influence on a field's ability to record information, in the form of constraints, then the student records that are recordable on Form 1 in Figure 7.1 are a subset of those student records that are recordable on Form 2. We employ terminology used to describe relationships among complex objects to describe the relationship between Form 1 and Form 2, referring to Form 1 as a *generalisation* of Form 2. Alternatively, we may say that Form 2 is a *specialisation* of Form 1.

To illustrate multi-form conflicts let us assume that a form such as the one in Figure 7.2 exists in the same interface as the form in Figure 7.1(a). Individually the two forms (7.1(a) and 7.2) can be seen as generalisations of the form in Figure 7.1(b). When the same two forms are considered as a single interface, however, they are equivalent to the the form in Figure 7.1(b). This can again be confirmed by examining the forms at the record level. The above example has illustrated how overlap conflicts may exist between interfaces comprising several forms. Overlap conflicts may also manifest themselves among interfaces where in one interface the fields that appear on one form are distributed across several forms in another interface, these are known as one-to-many overlap conflicts. Similarly fields spread across several forms in one interface may be spread across several forms in another interface, such that there is no 1:1 equivalence mapping between forms, this is called a many-to-many overlap in the two interfaces.

A further multi-form conflict occurs when either the columns of a combo box, grid or list differ between interfaces. Consider two interfaces which comprise a form

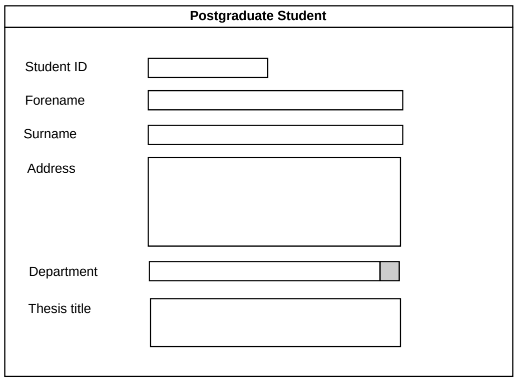


Figure 7.2: Postgraduate form

for recording a student's module grades. The grades are recorded in a grid and in one interface the grid contains the columns Module name and Grade. In the other interface the grid contains the columns Module code, Module name and Grade. Both interfaces also contain a second form, called Module Grade that is linked to the row of the grid. The module grade form records the module code, module name and grade in addition to providing more detailed information about the module, such as the number of credits the module is worth and the year and semester in which it runs. A conflict occurs here because it appears as though the Module name field is missing from the row of the grid in the first interface when it is in fact recorded on another form that is linked to the grid and which can be accessed by clicking a row of the grid. Such conflicts will be termed as missing subfield conflicts.

7.2.2.2 Field Conflicts

Conflicts existing among fields may be classified into representational conflicts and data conflicts. Representational conflicts are those which relate to the purpose of a field and its ability to capture information. Data conflicts pertain to the actual values recorded in the field. Examples of representational conflicts are differences in field names or differences in the types of control used to represent a field (see the

Student ID, Student # fields in Figure 7.1). The above representational conflicts are all examples of conflicts between single fields. It is also possible for representational conflicts to occur among several fields. To illustrate this consider how the two forms in Figure 7.1 represent a student's address. Assuming that postcode information is recorded in the address field in Figure 7.1(a) then the set of records recordable by this field is the same as the set of records recordable by the address and postcode fields in Figure 7.1(b). This is called a one-to-many multi-field conflict. Many-to-many multi-field conflicts are also possible: suppose that every house has a number and a House number field is added to the form in Figure 7.1(a). This results in the set of fields House number and address in Figure 7.1(a) recording the same information as the set of fields Address and postcode in Figure 7.1(b). Thus, a many-to-many multi-field conflict occurs.

Data conflicts arise among fields when the same information is represented using either: different domains; different units; or different precisions. Semantic differences such as these are part of a much wider problem and beyond the scope of this thesis. Approaches to dealing with this type of conflict are covered in (Kashyap and Sheth 1996), for example.

The complete taxonomy of differences among forms is shown in Figure 7.3. It should be noted that conflicts do not necessarily occur in isolation. For example, it is quite likely that form name, layout and missing field conflicts will occur alongside multi-form conflicts. Indeed, the previous example illustrated how missing field and form overlap conflicts can occur simultaneously.

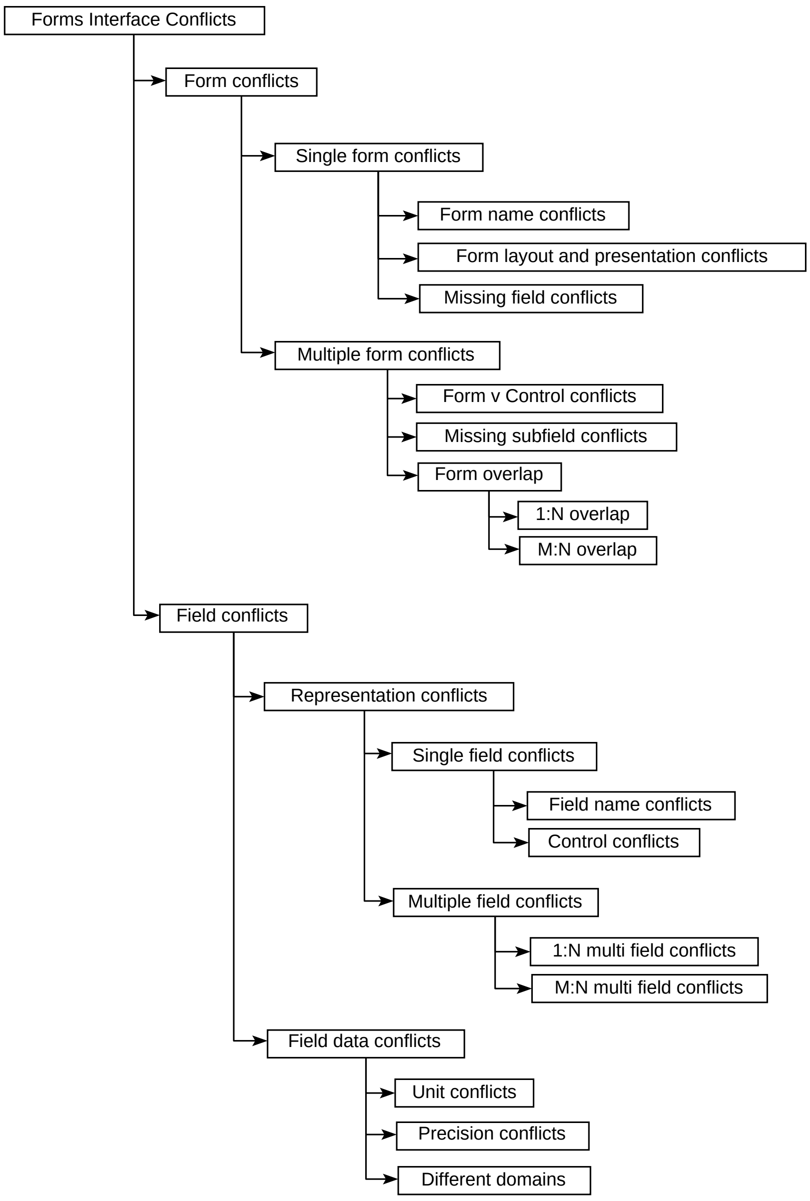


Figure 7.3: Taxonomy of differences between forms

7.2.3 Conflicts in a forms modification systems

In general, any of the conflicts identified in the previous section may occur between form interfaces. In this section we focus on the type of conflicts that are likely to occur in a forms modification system.

Recall from Chapter 1 that our forms modification system is based on a *drag-and-drop* paradigm that allows users to customise form interfaces by directly manipulating the fields on a form. Customisations are restricted to those that do not affect the schema of the database used to store information entered via the interface. Thus, a user cannot add a field to a form that would require the addition of an attribute to the database schema. It could be argued that by not allowing changes to the underlying schema our approach is overly prohibitive when compared to that of (King and Novak 1987) who allow schema modification. We feel, however, that our restriction is justified given that the target audience of our forms modification system is the novice user. A further reason for not allowing changes to an underlying database schema is that we make no assumption about the existence of an underlying database schema, and in practice one might not exist.

Given the constraint that no modification can be made to the database schema underlying an interface Table 7.1 lists the conflicts that may occur between form interfaces (indicated with a $\surd$). Those conflicts that do not occur in our forms modification system may occur between form interfaces in gerenal. If our approach is to deal with forms in a wider context then an appreciation of how these conflicts might be resolved is important. In the following sections we briefly consider how this might be performed.

The schema integration and multi-database literature provide a rich source of work that has considered the problem of identifying the same concepts in different

Conflict	**Can Occur**
Single-form	-
Form name	×
Form layout	√
Missing fields	×
Multi-form	-
Form vs. control	√
Missing subfield	√
Form overlap	√
single-field	-
Field name	×
Control	√
Multi-field	×
1:N conflicts	×
M:N conflicts	×

Table 7.1: Conflicts occurring in a forms modification system

database schemata thus resolving conflicts such as those discussed above. Early efforts in identifying similar concepts relied heavily on human intervention (see for example Sheth, Larson, Cornelio, and Navathe (1988)). Later work, however, addressed this problem with semi-automatic approaches being reported in the literature (see for example Ramesh and Ram (1995), and Gotthard, Lockemann, and Neufeld (1992)). Semi-automatic approaches reduce the amount of human intervention required, by generating lists of possible similar concepts and allowing the user to select those concepts which they deem to be equivalent.

7.2.3.1 Naming Conflicts

The resolution of many schema conflicts and the identification of similarities is based heavily on establishing relationships between the names used in different schemas. There are three basic relationships between names. These are:

- **Synonymy** - two distinct words are used to refer to the same concept;

- **Homonymy** - the same word is used to refer to two distinct concepts; and
- **Hypernymy** - one word is used to refer to a concept that is broader than the concept referred to by the other word.

Detecting these relationships between the names used in database schemata is the key to identifying similar concepts. Bright, Hurson, and Pakzad (1994) combine a thesarus and a dictionary in their approach to form a taxonomy of terms that includes both synonym and hypernym relations as well as definitions of the terms. Names from the database schema are mapped to entry-level terms in the taxonomy (leaf nodes) and the distance between pairs of names from different schema is calculated. The semantic distance metric (SDM) determines to what extent two names are classed as being similar. A pair of names with a low SDM are described as being *close* and, therefore, similar. A pair of terms whose SDM is high are seen as being dissimilar.

7.2.3.2 Structural Conflicts

Identifying concepts that are modelled using different structures has again received much attention from the database community. Several semi-automatic methods for the identification of inter-schema relationships have been reported in the literature (see for example Hayne and Ram (1990), Gotthard *et al.* (1992), and Ramesh and Ram (1995)) and provide potential solutions for both missing field and multi-field conflicts. The work of Ramesh and Ram (1995) for example could be applied to the problem of missing attributes. Their approach focusses on the use of constraint information as well as schematic infomation, such as functional descriptions of entity-types and attributes, to generate a set of inter-schema relationships. In a form interface information on constraints might be obtained by examining instances of *filled-in* forms, for example. Three types of inter-schema relationship are distinguished: equivalence, subsumption and overlap. Subsumption relationships would, for example, indicate missing fields whilst the overlap relation might prove useful in the detection of overlapping forms. Detecting multi-field conflicts may also be possible using techniques developed to identify equivalent, or component parts of, complex objects. Yu, Sun, Dao, and Keirsey (1991), for example, employ concept hierarchies as a means of detecting whether a collection of objects in one schema is the same as a composite object or collection of objects in another schema. The view integration system proposed by Gotthard *et al.* (1992) also recognised the need for a mechanism to detect components of aggregate structures, however, no details are given as to how this is implemented.

Unit and precision conflicts occur at the data rather than schematic level (Kim and Seo 1991). One mechansim for resolving such constraints is the definition of mapping functions between units. This would be a simple task for units where standard conversion functions could be used (Kashyap and Sheth 1996), however, for non-standard measurements, mappings would have to be specified manually. Where two fields use different precisions for their data, resolution can be more difficult and may be best left to the user.

Whilst a fully automatic approach to identifying all interschema relationships is not possible, user intervention could be substantially reduced by using the techniques discussed above. Any general approach to identifying equivalent structures between form interfaces, therefore, would have to be semi-automated, most probably requiring users to confirm or reject options presented to them. It is likely that the level of knowledge required to interact with a semi-automated system would preclude novice users from using the system. Such a system would, nonetheless, be useful to more advanced users.

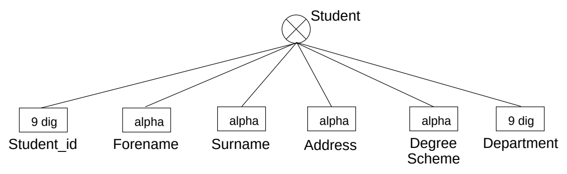


Figure 7.4: The IFO representation of the form in Figure 7.1(a)

7.3 Previous approaches to form equivalence

The equivalence of individual forms has also been touched upon, indirectly by the database community. Abiteboul and Hull (1988) whose work extends that carried out on the FORMAT model by Hull and Yap (1984) focus on the restructuring of hierarchic database objects for applications such as view construction and modification. By viewing forms as instances of hierarchic database objects Abiteboul and Hull describe how their work can be applied to forms. An IFO-like model is used to represent the hierarchic objects and comprises five main constructs: tuple ($\otimes$), set ($\circledast$), union ($\oplus$), basic ($\square$) and single-valued domain (1_f). Through the nesting of tuple, set and union constructs, basic and single-valued constructs can be combined to create hierarchic objects (see Chapter 5 for a complete review). Figure 7.4 illustrates the IFO equivalent of the student form in Figure 7.1(a).

Whilst the FIS and the IFO representations are similar, the IFO representation does not take into account the details of the controls used to represent fields. In this sense the IFO representation is less granular than our FIS representation as a single IFO construct represents several different interface controls whereas we have a one-to-one correspondence between CIOs in the FIS and interface controls. Whilst one could argue that more constructs means a geater risk of there being a conflict because there are more ways to model things, it is important to capture controls at this level of detail because even though several controls map to the same IFO construct they are not necessarily replaceable, as was shown in Chapter 5.

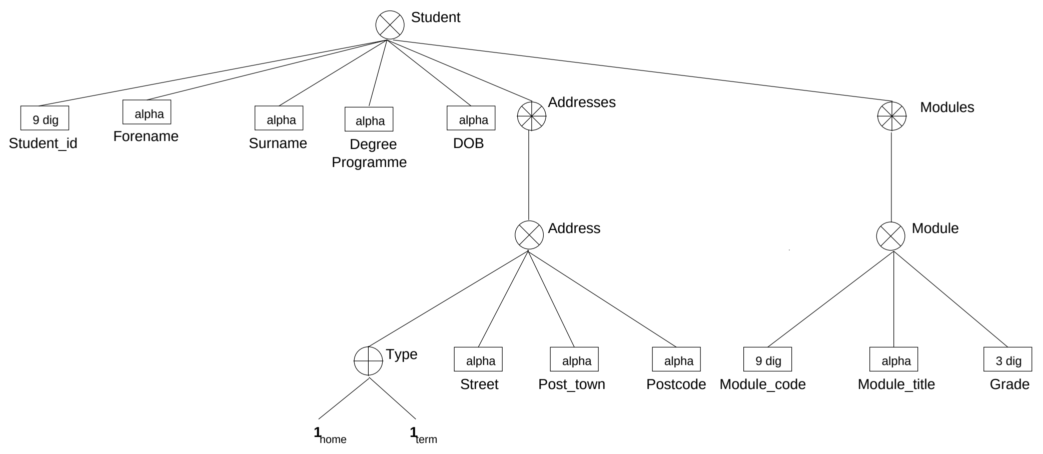


Figure 7.5: The IFO representation of a student form recording home and term time addresses

The most interesting aspect of Abiteboul and Hull's work is the algebra they have developed for manipulating the IFO structures. Using a pattern matching paradigm the algebra allows queries to be made on instances of IFO structures as well as the restructuring of the schemata themselves. Using the rewrite rules in Figure 7.6, for example, the IFO schema in Figure 7.5 can be transformed into the IFO schema in Figure 7.7. In addition to their algebra, Abiteboul and Hull present a series of restructurings which have been proven to preserve equivalence based on the definition of a normal form for IFO schemata. By defining a *normal form* for IFO schemata, it is shown how two IFO schemata in the same equivalence class can always be transformed, via the equivalence preserving restructurings, to the same normal form. Table 7.2 on page 153 shows nine of the equivalence preserving transformations grouped according to the construct they operate upon. The IFO schema in Figure 7.8, for example, can be transformed into the same normal form as the one in Figure 7.9 by applying the following sequence of transformations: simple $\times$ transformation on the *name* aggregation; rising $+$ transformation on the *type* union ; and renaming of the *type* union.

Whilst Abiteboul and Hull present some encouraging ideas to the problem of

$\rho_{home} =$	home_address:rew([street:x; post_town:y; postcode:z; type:$\mathbf{1}_{home}$] $\rightarrow$ [street:x; post_town:y; postcode:z])
$\rho_{term} =$	term_address:rew([street:x; post_town:y; postcode:z; type:$\mathbf{1}_{term}$] $\rightarrow$ [street:x; post_town:y; postcode:z])
$\rho =$	student:rew([student_id:s; fnames:t; sname:u; dob:v; degree_prog:w ; address:x; modules:y] $\rightarrow$ [student_id:s; fnames:t; sname:u; dob:v; degree_prog:w, $\rho_{home}(x)$, $\rho_{term}(x)$, modules:y])

Figure 7.6: the rewrite rules used to transform Figure 7.5 into Figure 7.7

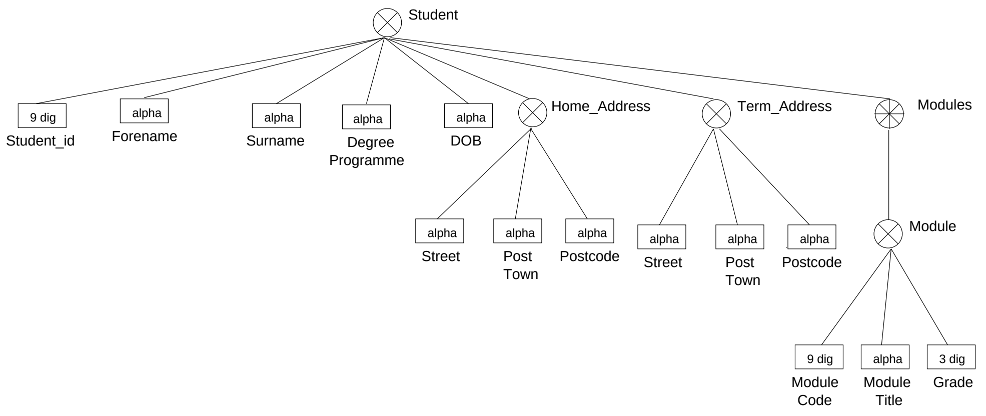


Figure 7.7: A restructured version of the IFO schema in Figure 7.5

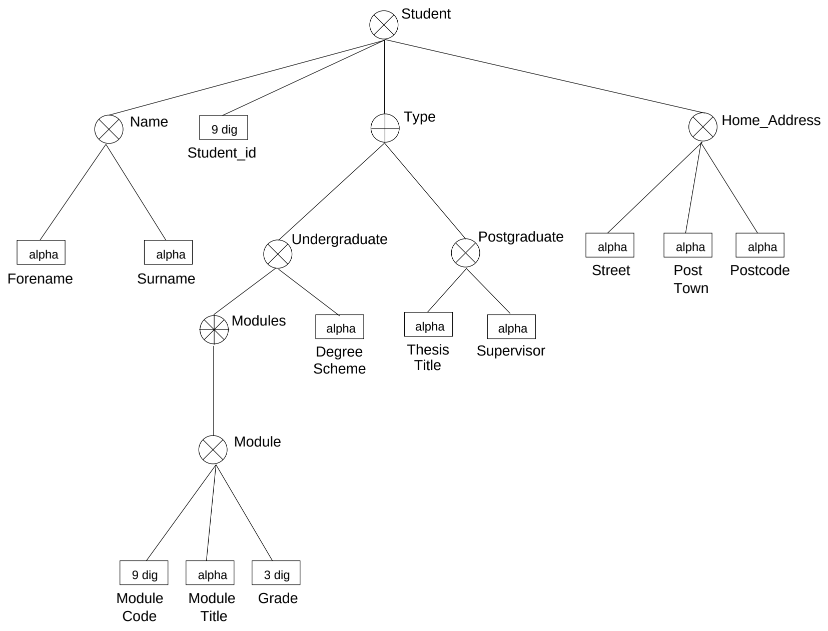


Figure 7.8: The IFO representation of a form capturing details of undergraduate and postgraduate students

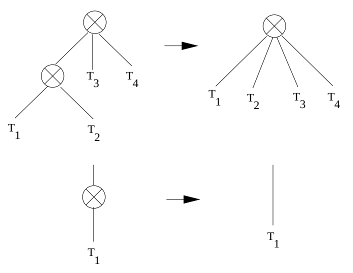
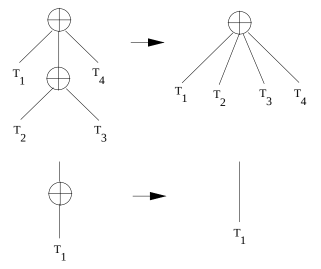
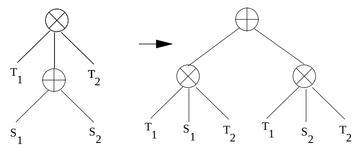
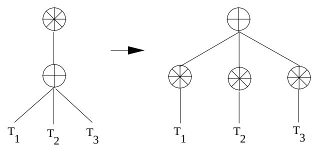
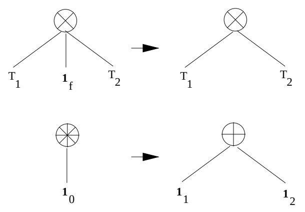
Type	**Transformation**
simple $\times$	
simple $+$	
rising $+$	
$*$	
simple $\mathbf{1}$	
renaming	$T_1 \rightarrow S_1$

Table 7.2: Equivalence preserving transformations for the IFO model, (Abiteboul and Hull 1988)

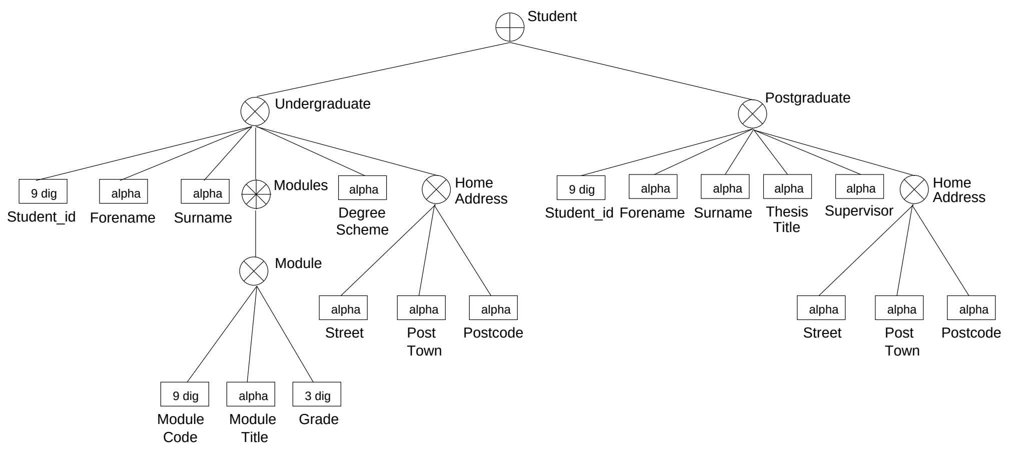


Figure 7.9: A restructured and equivalent version of the IFO schema in Figure 7.8

equivalence for graphical form interfaces, their approach lacks any mechanism to deal with the equivalence of several forms in which information on one form may be distributed across several other forms. Consider the example interface in Figure 7.10 that is equivalent to the single form interface shown in Figure 7.11. The lack of a *fusion* or join operator in the algebra that could be used to connect together IFO schemata via some common object results in the equivalence preserving restructurings being insuffiсent to prove the equivalence of the two interfaces. This motivates the need for mechanisms that allow the comparison of collections of forms.

One possible solution to the problem of multiple forms is presented by Guting *et al.* (1989) who propose an algebra for operating on the NST model discussed in Chapter 5. The nested sequence of tuples (NST) algebra accompanying the model provides facilities for the restructuring of NST schemas and is, therefore, capable of restructuring forms and creating new forms from existing forms. The NST algebra contains several operators including the familiar relational operators project, select and join. Here, however, we focus upon operators specific to hierarchical structures such as grouping (γ), distribution (δ) unpacking (μ) and the restructuring operator (λ).

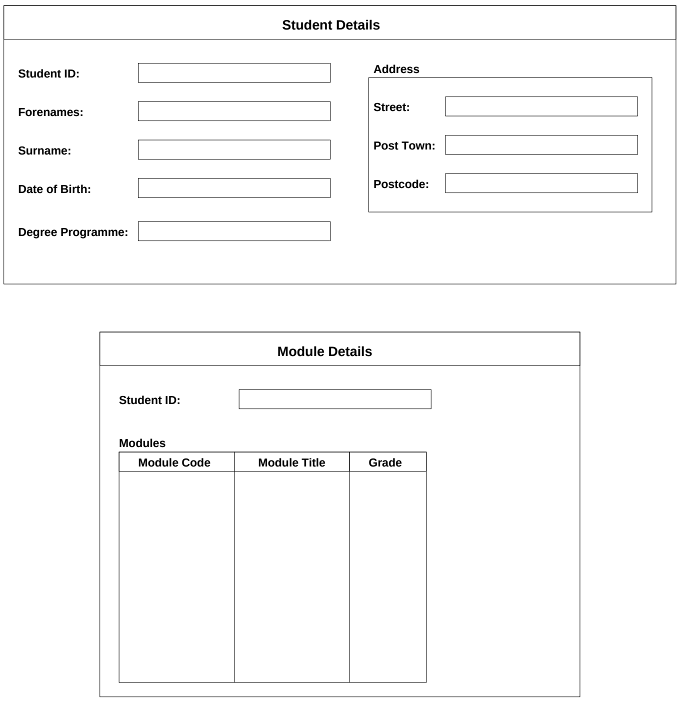


Figure 7.10: A multi-form interface for recording student information and details of the modules they are studying

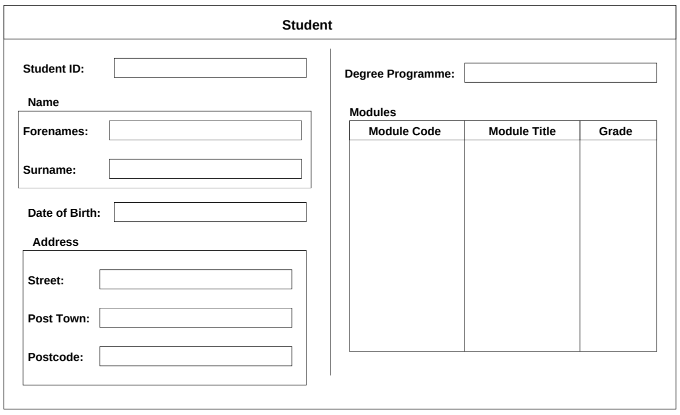
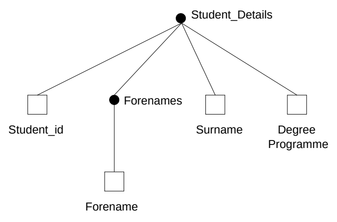
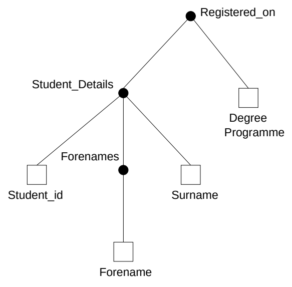


Figure 7.11: A single form interface recording the same information as the interface in Figure 7.10

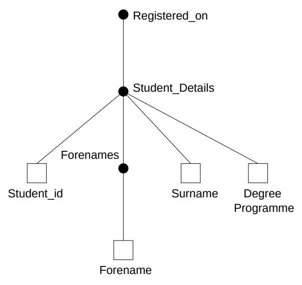
The grouping, distribution and unpacking operators are similar to the "nest" and "unnest" operations from the literature on non-first-normal-form relations. (Abiteboul and Bidot 1984) The grouping operator allows a collection of tuples to be decomposed into groups that share the same value in one or more atomic components. The result is a single hierarchical type for each group. Consider the student from in Figure 7.12(a). The application of the grouping expression γ[degree_prog{registered_on}] to the form in Figure 7.12(a) results in the creation of a sequence of tuples as shown in Figure 7.12(b). Each tuple comprising a degree programme and a sequence of students registered on the degree programme. The distribution operator (δ) takes the name of a composite component (e.g. registered_on) and an atomic component and replicates the atomic component in the composite component. The result of the operation δ[registered_on,degree_prog] on the structure in Figure 7.12(b) is shown in Figure 7.12(c). The unpack operator can be used to unpack a sequence of tuples in which each single tuple has a single component which is a sequence. The result is a sequence that is a concatenation of all the embedded sequences; thus removing one level of nesting. If the unpack operation, μ[students], is applied to the structure in Figure 7.12(c) then the result would be the structure shown in Figure 7.12(a). These three examples have shown how the grouping, distribution and unpacking operations can be used to restructure forms. Furthermore, in this particular example the restructurings have preserved the equivalence of the structures. It is possible to imagine a form with an underlying structure such as the one shown in Figure 7.12(b) which as a result of alterations by the user was modified to have the structure shown in Figure 7.12(a). Also if by applying the restructuring operators, distributon followed by unpacking, it was possible to arrive at the structure shown in figure 7.12(a) then we could say that, for this particular example, the changes had not lost any information capacity and the two forms are in fact equivalent.



(a) initial structure



(b) after grouping



(c) after distribution

Figure 7.12: An example of restructuring operators

The operations discussed in the previous paragraph were all limited to restructurings on a single form. The restructuring operator, however, is able to perform restructurings over several forms and is, in effect, the fusion operator missing from the IFO algebra. Using the restructuring operation objects can be constructed and embedded into existing objects; new objects can also be created through the combination of existing objects. The following restructuring operation may be applied to Figure 7.12(b): students λ[(registered_on **count**) {#_of_students}], for example. The count operation is used to count the number of students registered on a degree programme and the restructuring operator then adds this to the students tuple. The result is shown in Figure 7.13. The power of the restructuring operator comes when it is used over several forms. Consider, for example, the two forms shown in Figure 7.14 and their respective schemas in Figure 7.15. Using the restructuring expression in Figure 7.16 it is possible to combine the two forms into a single form. The restructuring expression takes the student_grades tuple and joins it with the student_details tuple using the student identifier. The projection operator then projects out all the objects except the s_id. The colon in the expression indicates that rather than adding the result of the operation to the existing structure (as in the previous example), it should replace it. This results in the form schema shown in Figure 7.18. The restructuring has combined the two forms into a single form which has the same schema as Figure 7.17, except for the title which can be changed through renaming. This example has illustrated how the restructuring operator can be used to peform restructurings which, in some cases, may preserve the equivalence of the forms. Using the restructuring operator, therefore, would enable collections of forms that potentially convey the same information as the single form to be compared.

Whilst the NST algebra supports the powerful restructuring operation the algebra alone is insufficient for the purpose of comparing schemata. The algebra does

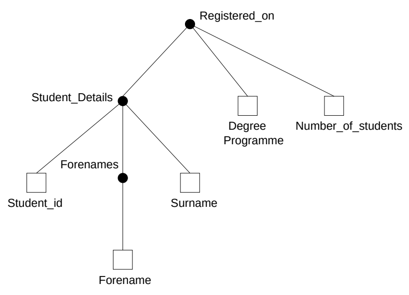


Figure 7.13: An example of the lambda operator

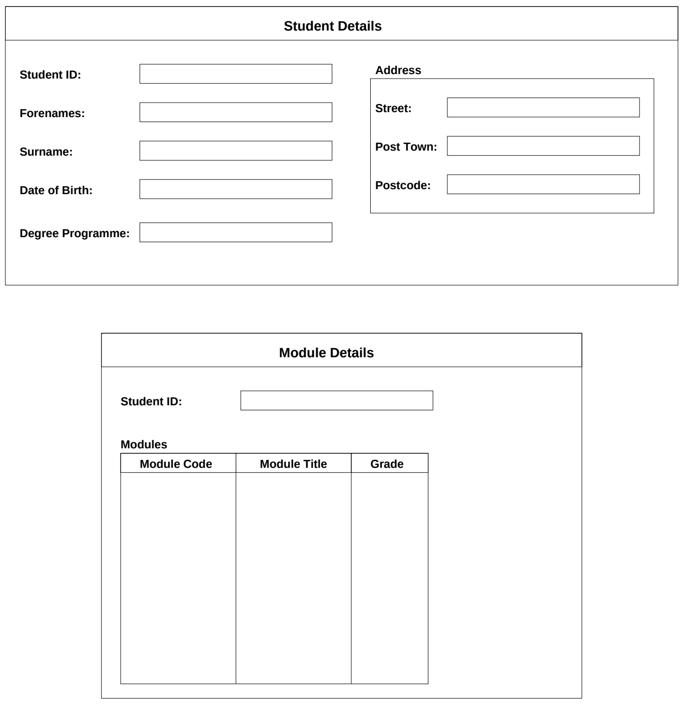


Figure 7.14: Student form split into two forms

not have the notion of a normal form, neither are any equivalence preserving transformations defined, without which one cannot guarantee that a restructuring has not led to a loss of information content. In addition to this a more fundamental problem exists which makes both the NST and IFO approaches unsuitable for the purpose of interface equivalence. Firstly, the non-trivial classification of interface controls as model constructs and secondly, hierarchic models are unable to represent interfaces containing cycles as was illustrated in Chapter 5.

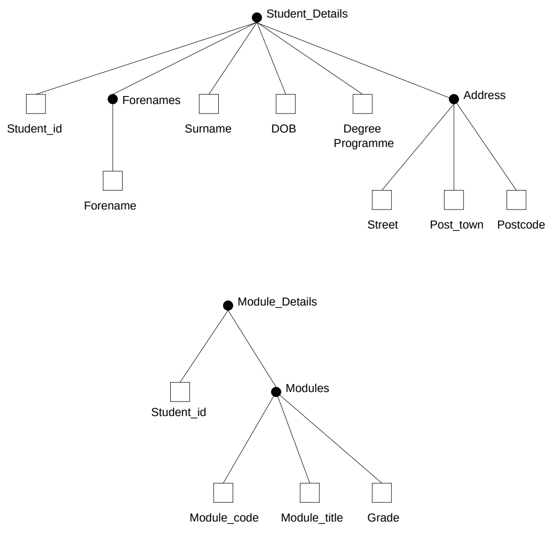


Figure 7.15: Type definition for the two student forms in Figure 7.14

student_grades
  λ[student_grades : student_grades student_detatils $\times$ [s_id = student_id]
 π[student_id, fnames, sname, dob, degree_prog, address, modules]]

Figure 7.16: restructuring operation

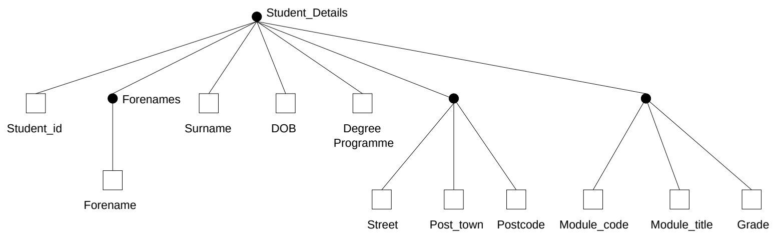


Figure 7.17: A form schema for Figure 5.1

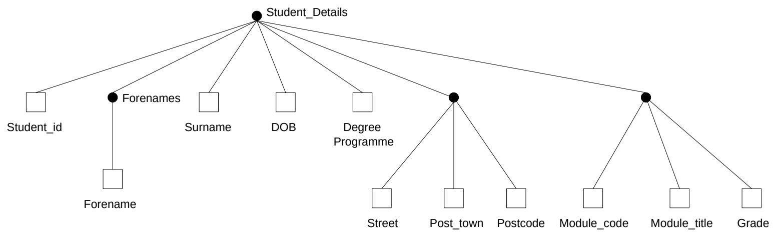


Figure 7.18: An example of the restructuring operator used to combine forms

7.4 An example form interface

To illustrate the equivalence metrics defined in the remainder of this Chapter we will use a small form interface comprising several forms. The interface is to a production planning system which belongs to the same suite of applications as those used in the case study in Appendix B. Figure 7.19 provides an overview of the forms in the interface and the links between the forms.

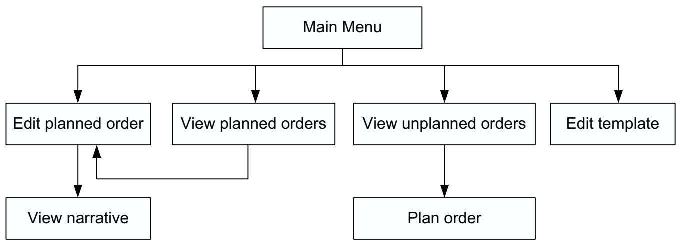


Figure 7.19: An overview of the planning application interface

For the purpose of the example several alterations will be made to the interface. These will be made to the Edit template form and will consist of changing the controls used to represent the fields on the form. As a result these changes will cause single-form conflicts to occur between the original and modified interfaces. This will allow us to demonstrate how equivalence can be tested in such situations. The more complex case of comparing interfaces exhibiting multi-form conflicts is dealt with in Appendix B.

7.5 A Framework for form interface equivalence

In the previous sections it was shown how other approaches to testing the equivalence of forms are insufficient for the purpose of comparing graphical form interfaces. This section presents a framework for testing the equivalence of graphical form interfaces that is able to deal with both single and multi-form interfaces. The framework employs both the form interface model from Chapter 4 and the transformations from Chapter 6. Central to our framework is the taxonomy of form interface conflicts, for it is the conflicts that occur between two interfaces that dictate the method of comparison.

Two types of equivalence are recognised in our framework. The first, and strictest definition, regards two interfaces, A and B, as equivalent if interface A conveys the same information as interface B and the interface controls used to convey the information are employed in accordance with user interface design guidelines. We call this definition of equivalence, maximum domain size (MDS) equivalence because, as will be shown in the following section, interface design guidelines, can be seen as imposing an upper limit on the size of the domain that can be represented by an interface control. The second type of equivalence differs from MDS equivalence in that it does not require interfaces to respect user interface design guidelines. Under this definition, two interfaces that convey the same information, but that do not respect user interface design guidelines can be identified as being equivalent. We call the second type of equivalence domain equivalence.

The rationale behind these definitions of equivalence is that when comparing interfaces any improper usage of controls, resulting from end-user modification, can be identified by testing the interfaces against both definitions. If a pair of interfaces are domain equivalent then we know they convey the same information. If, however, they are not MDS equivalent then there may be some error in the use of interface controls, possibly as a result of end-user modification, which can be identified and reported to the user to be corrected.

In the taxonomy of interface conflicts on page 145, a pair of form interfaces may suffer, at a form level, from either single-form or multiple-form conflicts. Single-form conflicts mean there is a one-to-one correspondence between forms in the two interfaces and that any conflicts can be resolved by examining pairs of forms in isolation. For the class of interfaces exhibiting these characteristics, it is possible to compare their form interface schemata, which, being directed graphs, can be achieved by testing for the existance of an isomorphism between the two graphs. Exactly how the test is performed is contingent upon the type of equivalence being tested. For domain equivalence it is sufficient to perform a pairwise comparison of the forms in the interfaces. For MDS equivalence it is necessary to decompose a form into a set of subschemata that correspond to the complex fields on a form. The subschemata of a form are then compared. In addition to comparing forms, it is also necessary to compare the relationships among forms (i.e. opens links in the FIS). Provided all the forms in a pair of interfaces are pairwise equivalent and connected in the same way then the interfaces are regarded as equivalent.

In interfaces exhibiting multiple-form conflicts, the correspondences among forms may be one-to-many and many-to-many in addition to one-to-one. For this class of interface, rather than comparing the forms interface schemata directly, the forms interface schemata are first tranformed into EER schemata. The EER schemata are subsequently compared using techniques such as those found in (Jajodia, Ng, and Springsteel 1983; McBrien and A. 1997; Knapp 1998). Alternatively, equivalence preserving restructurings could be applied to the EER schemata to map them into a “normal form” in a similar manner to the approach used by Abiteboul and Hull (1988). If this approach were used, it would not be necessary for the interfaces to have the same EER schemata, they would only need to share the same normal form, arrived at by applying a sequence of equivalence preserving transformations. The rationale behind the mapping of the forms interface schemata to EER schemata is that by describing the information conveyed by the interfaces in terms of entity-types, relationship-types and attributes the one-to-many and many-to-many correspondences that make it impossible to compare forms in isolation, are abstracted away. Whilst it is sufficient to compare EER schemata when testing for domain equivalence, an additional test is required for MDS equivalence. This additional test compares the controls used to represent the underlying attributes and entity-types to ensure they respect user interface design guidelines.

7.5.1 FIS Equivalence

The isomorphism test that will be used has two main requirements: an equivalence relation among the fields of an FIS; and an equivalence relation among the opens and contains relationships among fields. To define the former it is necessary to examine the controls that are used to represent fields. The classification of interface controls in Chapter 5 represents progress in this direction as it suggests that different interface controls are capable of representing the same types of object. In this section we expand the work of Chapter 5 and use the classification of controls to define the concept of interchangeability for fields.

7.5.1.1 Field replacement

Batini *et al.* (1986) describe two concepts as being equivalent if they are modelled by constructs that are equivalent in some data model. If the forms interface model is thought of as a data model for forms, and interface controls as the constructs of the model, then it is possible to observe relationships between its constructs. By classifying interface controls as constructs from the Generalized Semantic Model (GSM) (Hull and King 1987) it was shown in Rollinson and Roberts (1998) that several interface controls can represent the same GSM construct, suggesting the existence of classes of interface controls that are capable of being used interchangeably. Furthermore, an interface control can belong to several classes; the

Abstract Type	**Lexical Type**	**Grouping**
form	textbox	listbox
groupbox	column	grid
row	combobox_{single}	checkboxes
combobox_{multi}	radio buttons	
textbox	checkbox	

Table 7.3: Control to GSM Mapping

exact class being determined by neighbouring controls. For example, in one interface a combo box may have several columns, in which case it represents an abstract type, whilst in another interface a combo box might have one column and represent a lexical type. Table 7.3 provides an overview of the classification of interface controls from Rollinson and Roberts (1998). The existence of a control in a particular class, however, does not guarantee that the control is interchangeable with other controls in the class. Two factors influence a control's ability to replace another control, these are: (i) the maximum domain size of a control; and (ii) structural constraints.

The maximum domain size of a control is a limit on the set of domains that can be represented by a control. A control with a maximum domain size (MDS) of n can represent any domain whose size is less than or equal to n. The maximum domain size of a control is derived from user interface design guidelines. The MDS for a group of checkboxes or radio buttons is eight as guidelines suggest that no more than eight checkboxes/radio buttons should be used in a group (Galitz 1997). Textboxes, on the other hand, have no upper limit on their MDS because they can accept unconstrained input (Janssen, Weisbecker, and Ziegler 1993). List-based controls such as listboxes, grids and combo boxes have a recommended size of fifty items (Tullis 1985) whilst the spinbox has a limit of eight items in its list (Galitz 1997). The pseudo-controls, row and column, have no guidelines, indeed the MDS of a multi-column row is contingent upon the MDS of each of its columns. The MDS of the single-column row and the column control is taken to be the same as the textbox

Abstract Type	$S_1 = \{\ \text{form}(\infty)\ \}$ $S_2 = \{\ \text{combobox}(\infty)_{multi}\ \}$ $S_3 = \{\ \text{row}_{multi}\ \}$
Lexical Type	$S_4 = \{\ \text{textbox}(\infty), \text{combobox}(50)_{single}, \text{spinbox}(8), \text{radiobuttons}(8), \text{checkbox}(2)\ \}$ $S_5 = \{\ \text{row}(\infty)_{single}\ \}$ $S_6 = \{\ \text{column}(\infty), \text{combo}(50)_{single}\ \}$
Grouping	$S_7 = \{\ \text{grid}(50), \text{listbox}(50), \text{checkboxes}(8)\ \}$

Table 7.4: Control orderings

as they both, theoretically, allow unconstrained input. Maximum domain size can be considered a *soft constraint* on a control's ability to replace another. This is because it is based on guidelines, which may be relaxed, rather than concrete rules. Furthermore, the combo box control in some cases allows items to be added to its list which may lead to the limit of fifty items being exceeded. For this reason we allow the maximum domain size constraint to be relaxed. This provides two forms of the constraint. A strict form which adheres to the sizes dictated by the user interface guidelines and a weaker form which ignores the guidelines and allows all the controls in a class to be used interchangeably.

In contrast to maximum domain size, structural constraints are *hard constraints* on the interchangeability of controls. Structural constraints ensure that the structure of a particular form is consistent with that of the general model of a form in Chapter 4 and if relaxed there is a risk of illegally structured forms being created. Structural constraints have the effect of partitioning the classes of interface control into subclasses. Each subclass corresponds to the set of controls that can be used interchangeably. Table 7.4 shows a revised classification of controls in which the subclasses and the maximum domain size (in parentheses) are shown.

The relationships identified among controls can be used to provide a more useful indicator as to whether one field on a form is replaceable by another which considers not only the interface control but also the object represented by the control. By comparing the domain sizes of controls, the relationships between fields can be stated more precisely. Before examining the relationships among fields, it is important to differentiate between basic fields and complex fields. Recall from Chapter 4 that basic controls are those that appear as leaf nodes of the general model of a form whilst complex controls are those that appear as internal nodes (i.e. grid, listbox, multi-column combo box and multi-column row). A basic field is any field that is represented using a basic control and a complex field is any field that is represented by a complex control.

7.5.1.2 Basic field equivalence

Field equivalence between a pair of basic fields is determined by:

(i) their names; and

(ii) their expressiveness in representing objects

In the forms modification system, the names of fields will always be identical or be defined as synonyms via tables created during modification. To verify the expressiveness of a field it is necessary to: examine the domain of the object that the field represents; identify the extent to which this domain is represented by a field (i.e. does the field only represent a subset of the domain of an object); and identify the maximum domain size of the control used to represent the field. Thus, expressiveness is a control's ability to represent a particular object and its associated domain of values. A checkbox could be said to be expressive enough to represent the object *state* with the domain { on, off } but not expressive enough to represent the object *title* with the domain { Mr, Mrs, Miss }, for example. A field's expressiveness