Exploiting model-based techniques for user interfaces to databases

T.Griffiths[♠], J. McKirdy[♥], G.Forrester[♣], N.Paton[♠], J.Kennedy[♣], P.Barclay[♣], R.Cooper[♥], C.Goble[♠], P.Gray[♥]
[♠]Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, England. http://img.cs.man.ac.uk
[♥]Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow G12 8QQ, Scotland.
http://www.dcs.gla.ac.uk/ {~pdg | ~jo | ~rich }
[♣]Department of Computer Studies, Napier University, Canal Court, 42 Craiglockhart Ave, Edinburgh, EH14 1LT, Scotland.
http://www.dcs.napier.ac.uk/osg/

Abstract
Model-based systems provide methods for supporting the systematic and efficient development of application interfaces. This paper examines how model-based technologies can be exploited to develop user interfaces to databases. To this end five model-based systems, namely Adept, HUMANOID, Mastermind, TADEUS and DRIVE are discussed through the use of a unifying case study which allows the examination of the approaches followed by the different systems.

Keywords
Database interfaces, Model-based user-interface design environments, TADEUS, Mastermind, HUMANIOD, Adept, DRIVE

1 INTRODUCTION

Databases are ubiquitous, and almost every database application exploits interactive interfaces in some way, whether for database design, querying, data-entry or browsing. The importance of user interfaces to databases as a research topic is reflected in the various workshops and conferences that have been held in recent years ((Spaccapietra, 1995), (Kennedy, 1996), (Wierse, 1995)). The demand for interface builders for databases in commerce is reflected in the widespread use of form interface builders (Oracle, 1996) and toolkits with integrated database capabilities (Delphi, 1997). However, development of user interfaces to databases is still often ad-hoc, and there is little evidence of the most recent interface development environments, such as model-based systems, being applied to database interfaces.

Model-based interface development environments have emerged in recent years as a promising technique for supporting more systematic and more efficient user interface development (Foley, 1995), (Johnson, 1995), (Bodart, 1995), (Puerta, 1996). Not only do model-based systems support rapid prototyping through automatic generation of (preliminary) interfaces from partial descriptions of applications, they support a methodology that encourages discipline in interface design. However, model-based systems are not yet mature, and proposals differ significantly in the range and nature of the models supported. There are also few case studies of the use of such systems in practice, so it is difficult to compare and contrast different proposals.

This paper provides a case study on the use of model-based systems for database interface development. A simple library database application is described and its implementation presented using five representative model-based systems. The paper thus has the following aims:

- To illustrate to database interface developers the possible benefits of exploiting model-based techniques.
- To assess the appropriateness of current model-based systems for developing user interfaces to databases.
- To enable a comparison to be made of the facilities supported by model-based systems in a common application.

Although there have been earlier surveys of model-based systems (Schlungbaum, 1996), (Myers, 1995), (Szekely, 1994), these have never been in the context of user interfaces to databases, and have never exploited a case study to allow more detailed comparisons to be made of the different approaches.

The paper is structured as follows. Section 2 describes the case study used throughout the paper. Section 3 provides an overview of model-based systems. Section 4 describes the use of five model-based systems, namely Adept, Humanoid, Mastermind, TADEUS and DRIVE for supporting the case study. Section 5 summarises the capabilities of the different model-based systems, and section 6 presents some conclusions.

2 CASE STUDY

In section 1, reference was made to the scarcity of case studies which illustrate the use of model-based systems in practice and how this has made comparison of such systems difficult. In order that this paper can present a balanced survey of selected model-based systems, a small-scale case study has been devised and is described here. Thereafter, in section 3, each of the systems under comparison will be discussed with reference to their support of the case study.

This case study is based upon a library database, the ODMG (Cattel, 1996) schema for which is shown in figure 1. Object classes are shown as rectangles and scalar types as ovals. In section 4 access to the database from the model-based systems is assumed to be through standard ODMG call interfaces.

Rather than attempt to describe an entire library application based on this domain, the case study focuses on a single task - S*earching for a book*. The case

study is not prescriptive with respect to interface style since to be so might unfairly influence the comparison of the various systems.

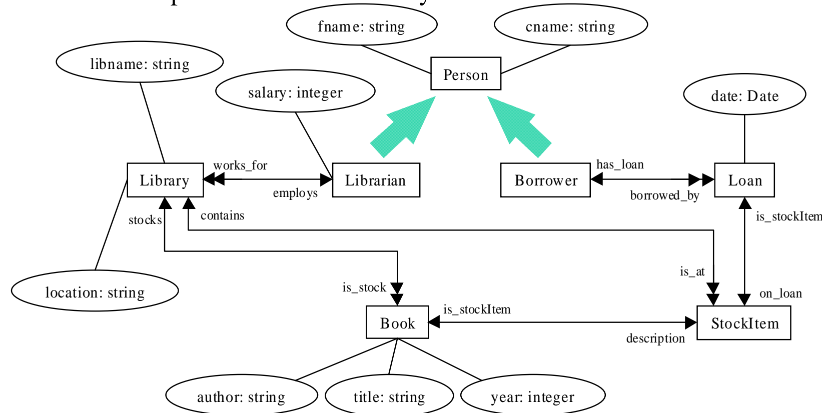


Figure 1: The Library Schema

To search for a book, the user must first specify that a search is to be performed. A series of search parameters must then be specified, including: the attribute(s) of the book on which the search is to be based - i.e. author, title or year; and whether exact or approximate results should be returned. By specifying these parameters, the user has effectively constructed a query that can then be submitted to the database. The system must either present the results of the query to the user in a manner appropriate to the particular interface style, or inform the user that no matching books were found. Where results were obtained, the user can browse through them.

3 OVERVIEW OF MODEL-BASED SYSTEMS

Model-based interface development environments (MB-IDEs) can be classified as IDEs that exhibit three main features, namely they: Support the **automatic generation** of interfaces; utilise **declarative methods** to specify the interface; and adopt a **methodology** to support the development of the interface.

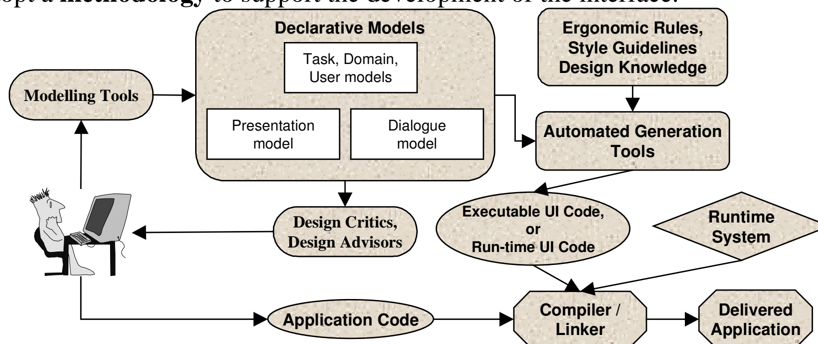


Figure 2: Typical Model-based Development Life-Cycle

The process of developing a user interface with a MB-IDE is an iterative process of developing and refining a set of declarative models using graphical editor tools or high-level specification languages. Once developed, these models are transformed according to ergonomic rules and/or style guidelines into an interface specification. This specification is subsequently linked with the underlying application code to generate a running application. This development process is illustrated in figure 2.

From the designer's viewpoint the key components of a MB-IDE are the declarative models which store the conceptual representation of the interface. During recent years, the models supported by MB-IDEs have increased both in number and in expressiveness. The first-generation of MB-IDEs (typified by UIDE (Foley, 1995), MECANO (Puerta, 1996), and JANUS (Balzert, 1996)) concentrated on modelling the underlying application domain using a **domain model**. This model represents the *application objects* together with their *properties*, available *actions*, and any *pre- or post-conditions* applicable to these actions. Typically, such a limited view of the modelled domain produced simple menu or forms-based interfaces.

In recent years however, MB-IDEs have exploited a much wider range of interacting models, with a consequent increase in the quality and variety of the generated interfaces. These models allow the designer to exploit fully the information gathered during the requirements analysis phase of the development process.

A **task model** allows the hierarchical description of the *tasks* performed by the end-user, including the ordering of sub-tasks, their *goals*, and the *procedures* used to achieve the goals. These procedures represent application-level operations, and are thus captured in the domain model. The development of the task and domain models are therefore closely related activities.

A **user model** describes characteristics of the intended users or groups of users of the application that can be exploited to tailor the functionality or the appearance of the resulting interface. Through the use of a user model a MB-IDE can generate different interfaces for each category of user, thus allowing the capture of both application independent (e.g., user capabilities, psycho-motor skills) and dependent (e.g., system knowledge, privileges) characteristics.

To allow the realisation of the dynamic behaviour of an interface a **dialogue model** can be provided. This model describes the interaction between the human and computer in terms of when and what commands can be invoked in a presentation independent manner. Earlier MB-IDEs derived this information from their task or domain models. The provision of an explicit dialogue model results in richer interfaces that more closely reflect the desires of the designer.

The provision of a **presentation model** allows the designer to specify the characteristics of the interface components. This can apply to both the static (widgets etc) and dynamic (typically involving run-time application-dependent data) facets of the interface.

Some MB-IDEs (e.g. Trident (Bodart, 1995)) also try to help the designer by providing design critics or advisors. These are tools that analyse models and either

suggest improvements or identify inconsistencies or errors in the design. In addition to these facilities other systems (e.g. UIDE (Foley, 1995), HUMANOID (Luo, 1993)) utilise the specified models to automatically generate non-interface features such as help and redo/undo sub-systems.

3 SURVEY OF SYSTEMS

3.1 ADEPT

Adept (Johnson, 1995) is a MB-IDE that follows a user centred design philosophy. To this end Adept utilises the user and task descriptions captured during the requirements analysis phase of application development to produce an executable user interface corresponding to the modelled tasks. Adept also recognises the importance of incorporating the proposed users of the developed system at all stages of interface development by involving the end-users during each stage of model development.

User Model

The Adept user model (Kelly, 1992) takes the form of a rule-base which is used to describe groups of current/proposed users in terms of both concrete attributes such as their domain knowledge and familiarity with computer systems, and abstract notions such as motivation and attitude. Once elicited, this knowledge can be used to provide design guidelines, and hence to influence the form of the generated interface. For example, in our case study two user groups can be identified, namely *Borrowers* and *Librarians*. A knowledge base is instantiated for each of these user groups. The Librarians user model will contain facts such as: ('application experience' 'high'); ('frequency of use' 'high'); ('motivation' 'moderate');

The individual user models are used in conjunction with a set of design rules that associate the modelled user characteristics with a set of interface design rules. For example the rule: ('typing skills' 'low' 'fill-in forms'); states that if the user group has low typing skills then fill-in forms might be an appropriate medium for the developed interface.

Task Model

The task of producing a UI to represent the modelled domain is a process of successively refining the tasks informally identified during the initial requirements analysis phase. The first stage in this process results in a refinement of the initial task model called the *task knowledge structure* (TKS) (Johnson, 1991).

The TKS is a representation of the tasks in a domain. Each task is modelled in terms of its *goals* (the state to be achieved by the task), *procedures* (the sequence of actions which will achieve the goal), *actions* (lowest level of activity), and the *objects* (as identified in modelled domain) which the task will affect. Adept does not make formal reference to an explicit domain model, rather the affected objects are listed with a textual description of their properties. For example the *Check in Book* task will affect the domain objects *book* and *borrower*, amongst others. The limited role assigned to the domain model means that it is less obvious how a database query could be constructed or its results processed using Adept as it is.

Figure 3 illustrates the top-level TKS for the case study. Nodes in this task model represent either goals, sub-goals, procedures or actions, with a temporal relationship specified between nodes. This relationship may be one of: *sequence* (whose order is inferred top-down); *interleaved* (multi-threading); *parallel*; or *choice*. Goals can also be specified as being *repeated*, *optional*, or *disabled* (i.e. not possible under certain conditions). The TKS also displays and allows the editing of the domain objects affected by each task. Unexpanded goals are indicated by the dark triangle.

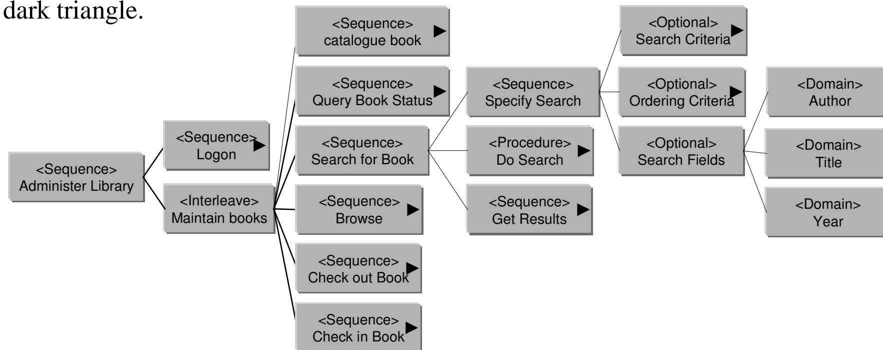


Figure 3: Case study TKS

Whilst the TKS seeks to describe *all* tasks from the users' perspective, the delivered system may only represent some of these tasks (e.g., the library system will not catalogue new books). The *Abstract Interface Model* (AIM), which is derived from the TKS, provides a high-level presentation independent specification of what the interface is to do.

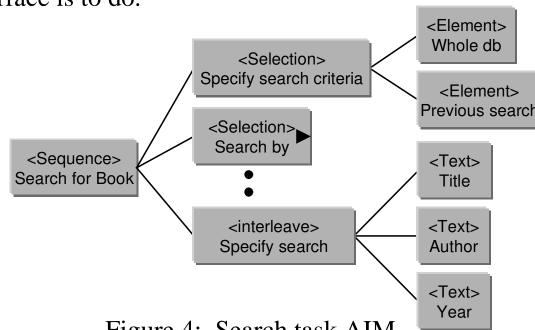


Figure 4: Search task AIM

The AIM represents the abstract interaction objects (AIOs) that comprise the user interface. Each AIO is modelled only in terms of the type of input expected. This type is provided by the developer form a set of pre-defined alternatives including text, number, range, element, set, or special, thus providing a link to the underlying type of the domain. Temporal information is also expressed about the AIM components, leading to a basic dialogue description of the abstract interface. The AIM corresponding to the Search task hierarchy is shown in figure 4.

Generating the User Interface

The high-level description of the user interface defined by the AIM must be transformed into a low-level platform independent description of the designed interface. This low-level description is provided by the *Concrete Interface Model* (CIM). Adept uses the Smalltalk language to produce an equivalent executable version of the interface.

The CIM is automatically generated from the information stored in the AIM and the user model specifications by instantiating each AIO as a widget from the adopted widget set. This process is achieved by the CIM interactively interrogating the user models' knowledge base of applicable design rules. For example, when the CIM needs to instantiate the widgets corresponding to the 'Specify search criteria' goal in the case study AIM, the choices radio button, checkbox, or button could be applicable. If the CIM generator cannot resolve a conflict then the designer is brought into the process through an interactive dialogue with the generator.

Methodology

Figure 5 provides an overview of the methodology followed by Adept. This is a semi-automated iterative process that is initiated by the development of user and task models. Whilst the current implementation of Adept does not allow for iterative refinement of the models, the authors state that this is one of the goals of the system.

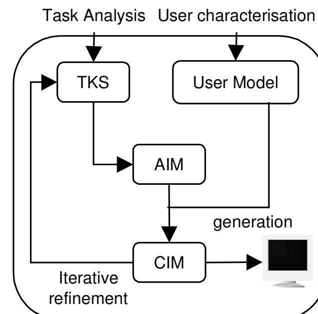


Figure 5: The Adept Interface Development Methodology

3.2 HUMANOID

HUMANOID (Luo, 1993), is a template-centred, MB-UIDE which aims to bridge the gap between interface builders and automatic interface generators. In contrast to the latter, HUMANOID was developed on the basis that the most difficult design decisions are best left to the designer since they require knowledge which is more easily and quickly modelled by humans than by a system, for example knowledge about end-users and the application domain.

A HUMANOID model comprises five semi-independent dimensions: *application semantics*, *presentation*, *manipulation*, *sequencing*, and *action side effects*.

Application Semantics

This model corresponds quite closely to the domain model described in section 3, and is designed independently of the manner in which the objects will be

displayed. Application objects and commands are hierarchically modelled by specifying the types and slots of each object. Commands are further modelled in terms of their inputs, preconditions, and a call-back procedure. Commands and objects can be grouped into application objects, multiple instances of which can be created at run-time. HUMANOID also facilitates the specification of data flow constraints.

The Application Model component of Figure 6 shows how this might be applied to the case study. The search task is modelled in terms of the input parameters for the query, the commands the user can invoke, and the grouping of these commands. The search command which refers to the submission of the query to the database, is itself specified in terms of its input parameters. Unfortunately, no details were available concerning the way in which the call to the database could be done and how the results could be made available.

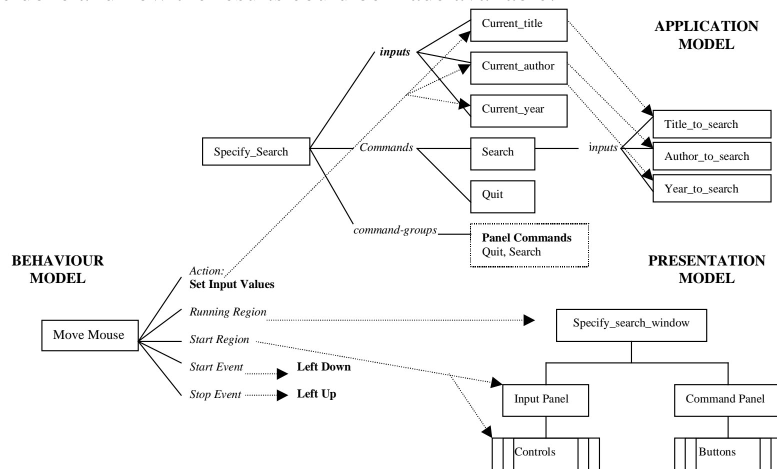


Figure 6: The Case Study as Specified According to HUMANOID

Presentation Model

This model relates directly to the presentation model discussed in section 3, and is specified using templates organised into a hierarchy such that the leaves consist of graphical primitives or primitive building blocks of the underlying toolkit. The information that defines a presentation template is represented as slots in the presentation template object. Such information includes: input data, applicability conditions, widgets and associated parameters, and component parts. HUMANOID provides libraries of templates that the developer can use directly or can extend.

The presentation model component of figure 6 outlines one possible interface for the search task and specifies that the interface comprises an input panel which has a number of controls (for allocating values to the query parameters) and a command panel containing the buttons for submitting the search query and for quitting the query.

Manipulation
Modelling manipulation involves stipulating the input gestures that users can perform to manipulate presented information plus the actions which should be invoked upon gesture detection. This is achieved by adding behaviour specifications to templates. In essence this model groups information that is contained within the presentation and dialogue models of other systems. A manipulation model includes specification of the gesture, where in the presentation it applies, the application data on which it operates, and actions to be taken at points during the gesture (e.g., setting a value). Manipulation specification for the case study is shown in the behaviour component of Figure 6. This demonstrates the connection between the presentation and the application model.

Sequencing
This model corresponds to the dialogue model described in section 3, and involves specifying the order in which displays appear on the screen plus the set of behaviours that are enabled at any instant. Designers do not directly state when commands are enabled or disabled in their design. Instead, the run-time system calculates timeous sets of enabled behaviours based on specified constraints. Like the other HUMANOID models, sequencing is modelled through properties of command, input and group objects. A library of objects is provided that implement commonly-used sequencing features. Unfortunately, insufficient documented information was available to enable the construction of a sequencing model for the case study.

Action Side Effects
HUMANOID also provides libraries of objects for commonly used side effects. These include: Beep-When-Correct, Message-When-Correct, etc. Other systems would perhaps incorporate these features in the dialogue model. Again, insufficient information made it impossible to reflect the action side effects as applied to the case study.

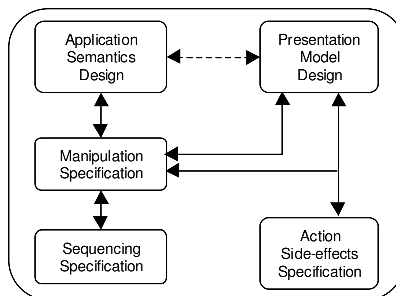


Figure 7. The HUMANOID Interface Development Methodology

Methodology
Figure 7 shows that HUMANOID's methodology would appear to be iterative, sequential, and partially automated (with some degree of concurrency). Specification of the application semantics is pivotal to user interface development

and can run in parallel with the presentation model design. Manipulation specification links application semantics and the presentation model and so is dependent upon them. Sequencing specification is entirely dependent on the manipulations. Action side-effects link manipulations with their side-effect on the presentation.

3.3 MASTERMIND

MASTERMIND (Szekely, 1996) is an MB-IDE based on two existing MB-IDEs, namely HUMANOID (Luo, 1993) and UIDE (Foley, 1995). It attempts to incorporate the strengths of both its predecessors while avoiding many of their weaknesses.

The MASTERMIND project has identified the need for three distinct models, namely the application, task and the presentation models, each of which is implemented using CORBA IDL (OMG, 1995) and operates as a "model server" process to which all the tools connect. While the models are separate and distinct, communication between them is facilitated through the use of an expression language.

Task model

The task model is used to specify what the user does with the interface, and drives the operation of the interface. The designer describes this by modelling high-level user tasks as compositions of necessary sub-tasks. An example of a task–subtask relation is shown in figure 8.

```
Search : Task {
  goal = "To search for a book in a given locus to a specified degree of accuracy";
  task_type = User;
  parameters = searchlocus : Parameter {type = String;},
               searchattribute : Parameter {type = String;},
               searchvalue : Parameter {type = String;},
               searchaccuracy : Parameter {type = String;};
  subtasks = :Task Connection {
  connection_type = SEQUENCE;
  tasks = Invoke_Search, Specify_Attribute, Specify_Value, Specify_Accuracy,
               Perform_Search; } ;
  is_reentrant = TRUE;
  is_interruptable = TRUE;
};

Specify_Attribute : Task {
  goal  = "Indicate the value to search the given attribute for";
  task_type = Interaction_Technique;
  task_extension = : Technique_extension {
  interactor =: Am_Text_Edit_Interactor {
  object = Search_Value_Field
  ……
  effects = [searchvalue <- Search_Value_Field.contents];
};
```

Figure 8: Case study Task Model Sub-tree

A task object has a type that determines its nature. For example, there exist task types to specify low-level interactions from the user (e.g. button click), and requests to display information or to perform some application processing. Tasks have goals and effects that are defined as expressions for evaluation. The effect of a task may be on the task model itself, the application or the presentation.

In the figure 8, the *Search* task is defined as a sequence of sub-tasks which must be performed. The sub-task requires user interaction as defined by its type *Interaction_Technique*. The use of *task_extensions* links the task model to the presentation model. This is demonstrated in the *Specify_Attribute* subtask by the setting of the *searchvalue* task object based on the value of a presentation object.

Application model

This model defines the classes that are used within the interface that represent real-world artefacts, and provides a mechanism for defining useful data types. Class structure and method signatures are both represented. The application model extends IDL to include notions such as preconditions on the execution of methods and the support for a publish/subscribe event model. The event model extensions allow task and presentation objects to register interest in an artefact and be informed should any changes to it occur.

Presentation model

The presentation model is used to describe how the interface appears to the user. Presentation objects are used to represent the static elements (windows, buttons, etc.) of the user-interface and the visualisation of the application data being manipulated. Presentations are typically defined as compositions of smaller presentation objects. Each presentation object is a specialisation of a prototype presentation object e.g. an interface component may inherit from a button widget. Furthermore, presentation parameters may be specified that determine the data displayed by the presentation object and the appearance of the presentation object itself e.g. colour, orientation.

```
DB_Interface : Window {
parameters = results : Parameter {Value = DB_Task.results}, font : Parameter {…..};
guides =    hguide1 : Guide { direction = HORIZONTAL; position = 200;},    // top of results box

            hguide2 : Guide { direction = HORIZONTAL; position = [(hguide1+bottom) /2];},
                        …….
grids = resultsGrid : Grid {direction = HORIZONTAL; start= [hguide1]; end= [hguide2];
                                        stretchable= FALSE; distance= [ font.height() +2 ];};
parts = bookdetail : BookDetail_Presentation {
                                    replication= { is_on_demand= FALSE;
                                    replication_data= [results.contents() ];
                                    references= grid_ref {reference = [resultsGrid];};
                                    };
                        ……..
};
```

Figure 9: Case Study Presentation Model of Application Main Window

MASTERMIND's presentation model differs from many other MB-IDEs in its attempt to provide a mechanism for specifying both the dynamic and static

components of the interface. The lack of support for the dynamic aspects of user-interfaces was identified by Szekely as the "Main Window" problem during the earlier HUMANOID project (Luo, 1993). Some forms of interface allow direct and indirect manipulation of a visualisation of application data. The layout and number of constituents of this visualisation may change at run-time and therefore cannot be specified at design time. The presentation model provides guides, grids and conditions to allow such a dynamic display to be defined abstractly.

The definition in figure 9 shows how a main window providing the search functionality could be described in MASTERMIND. The use of guides and grids is demonstrated showing that the positioning of interface elements is logical rather than physical.

Database Integration with MASTERMIND

The data model of an ODMG-compliant database and MASTERMIND's CORBA IDL-based domain model are conceptually similar in structure. It is therefore possible to wrap an ODMG database in CORBA IDL so that MASTERMIND's domain model can interface with the database. Once defined, method bodies can delegate execution to the equivalent C++ bindings of the ODMG classes in the database. Some code may be required to open and close the database.

Methodology

MASTERMIND's methodology assumes that early design documents have been prepared and begins with the specification of the initial three models. These are instantiated from early design artefacts. The specification of these models may be undertaken in parallel but some co-ordination will be required. From the documentation available these models are defined using a language interface only. No reference is made to graphical notations or tools to support them.

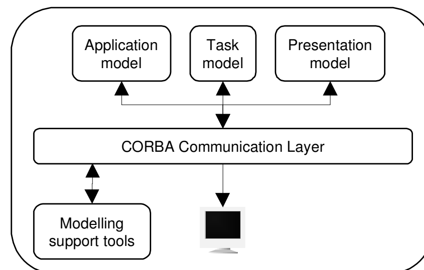


Figure 10: The Mastermind Interface Development Methodology

From the first three models a small procedural program analyses the models and generates the appropriate code to implement the user-interface. During code generation the declarative IDL models are encoded into C++ for compilation to provide an efficient run-time. However, the compiled applications retain the ability to access the original models by contacting the model server. It is possible for MASTERMIND to automatically generate sections of the interface according to a set of rules built into the program. Executable interfaces can be generated even when the models are not fully specified.

3.4 TADEUS

The TADEUS MB-IDE (Elwert, 1995) utilises four explicit declarative models, namely task, domain, user and dialogue models. The methodology proposed by TADEUS consists of both manual, computer-assisted, and automatic stages of development, the output of which is an interface specification file suitable for use by an existing user interface management system (UIMS) (TADEUS uses the ISA Dialog Manager (ISA, 1995)).

Task Model

The developed task model is realised as a hierarchical structure of user goals with a similar structure to Adept's TKS (Johnson, 1991). Each modelled goal consists of a *task*, *roles* describing the user groups interested in performing the task, and *domain objects* identified from the domain model which provide the primitive domain-specific functionality of each task. All goals and sub-goals have an explicit temporal ordering applied to them. The task model for the case study corresponds to the TKS developed in the Adept section of this paper.

Domain Model

TADEUS's domain model is an object-oriented realisation of the application domain using modelling techniques such as those of OMT (Rumbaugh, 1991). There is an explicit relationship between the task and domain models, with each task specifying the objects, attributes or methods that are utilised in the execution of the task.

User Model

The user model describes potential or existing groups of users in the modelled domain in terms of their roles and relations to specified tasks. Roles are described hierarchically in terms of task independent and task dependent attributes; for example, the user's level of experience with interactive systems. The specific tasks that each role performs are modelled through a *usage relation* that has attributes to represent concepts such as frequency of use, preferred input device, etc. The authors do not expand on functionality of the user model or how the model affects the generated interface.

Dialogue Model

Once the task, user, and domain models have been constructed, the first step towards producing the dialogue model requires the designer to specify *views* in the developed task or domain models. A view represents related processing units that should be simultaneously presented in a window in the target UIMS. The designer specifies the required views by annotating the task model to indicate which groups of goals should form each view. Figure 11 represents some of the views identified in the task model of the case study.

Dialogue graphs

The annotated task/domain model is used to automatically generate the initial state of the corresponding dialogue model using a notation called *dialogue graphs* (Schlungbaum, 1996a) to specify the dynamics of the interface. These graphs allow

the realisation of multiple instances of windows, hierarchical dialogue structuring, and the declaration of modal dialogue windows. A dialogue graph further distinguishes between *navigation dialogue* (the sequencing between views, realised via dialogue graphs) and *processing dialogue* (the dialogue within a view, specified through interaction tables).

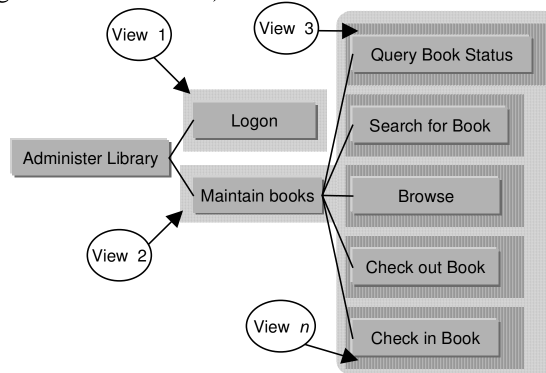


Figure 11: Annotated Task Model

Taking as input the automatically generated dialogue graph, the designer proceeds by editing this graph to model the required dynamics of the navigation dialogue. The dialogue graph corresponding to the case study is shown in figure 12.

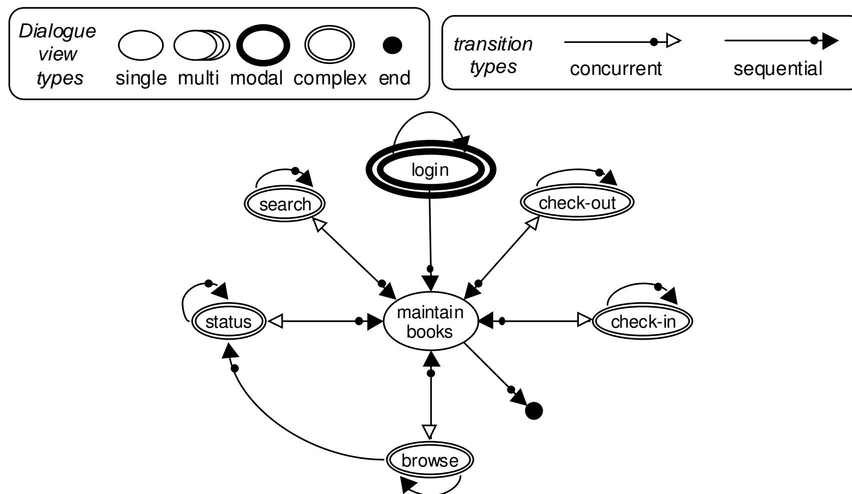


Figure 12: TADEUS Dialogue Graph and Legend

Figure 12 indicates that interaction with the developed library system will be initiated through a modal dialogue with a login window. Upon successful login, the user then has the choice of five possibly concurrent iterative book maintenance processes, each of which concludes by returning to the maintain books window.

The complexity of the graph is reduced through an information hiding mechanism, used here to hide details of the top-level groups. For reasons of brevity, figure 12 only shows one end node, although the case study requires navigation to end from each of the identified views.

Interface Generation

Initially the designer must specify the default layout description for each UI project. This consists of specifying some of the layout properties of interface objects, for example: All windows will have a white background and will use an arrow cursor.

For each identified dialogue view the designer must define the mapping for each form of dialogue to an AIO, and hence to its concrete interaction object (CIO) equivalent; an example is shown in tables 1 and 2.

Dialogue form	*type*	*AIO*
data input	free	input field
	1:m	single selector
	m:n	multiple selector

Table 1: Mapping dialogue forms to AIOs

AIO	*type*	*CIO*
input field	free	edit text
single selector	1:m (m=const, n ≤ 7)	group box + radio buttons
	1:m (m=const, n > 7)	list box
multiple selector	m:n (m=const, n ≤ 7)	group box + check boxes
	m:n (m=const, n > 7)	list box

Table 2: Mapping AIOs to CIOs

Interaction Tables

For each identified view the designer must define the processing dialogue via interaction tables, as illustrated by table 3 for the S*pecify search* task view. Table 3 identifies two groups in the corresponding CIO window and specifies the dialogue form for each grouped element. The mapping for each dialogue form to CIO is achieved through the rules defined in tables 1 and 2.

Transition	*dialogue form*	*type*	*group*	*group position*
title	data input	free	1	1
author	data input	free	1	2
year	data input	free	1	3
exit	function call	N/A	2	1
ok	function call	N/A	2	2

Table 3: Mapping AIOs to CIOs

Database Integration with TADEUS

There is a conceptually simple mapping from TADEUS's object-oriented domain model (using UML or similar) to the ODMG realisation of the case study. With the exception of stating that object methods form the substance of tasks, it is not clear how TADEUS would interface to an external system such as a database. It can

however be envisaged that such a linkage can be achieved through the database API methods by wrapping the required database functionality.

Methodology

The general stages in the TADEUS methodology are outlined in figure 13. It can be seen that there are three broad stages involved in the generation of an interface, namely requirements analysis, dialogue design and definition of default layout characteristics, and interface generation. No iteration is possible between each stage.

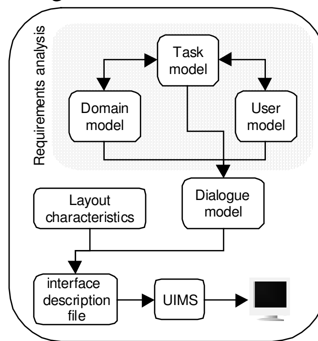


Figure13: The TADEUS Interface Development Methodology

3.5 DRIVE

DRIVE (Mitchell, 1995) is a MBS explicitly aimed at producing interfaces to databases rather than interfaces to applications in general. DRIVE combines interface modelling using a declarative language with storage of interface definitions in a database along with the enterprise data. DRIVE does not attempt to explicitly identify distinct models with different roles, rather it presents a framework for defining interfaces to database systems (IDSs) (Mitchell, 1996a), shown in figure 14. Components of this framework correspond to disparate models used by other MB-IDEs.

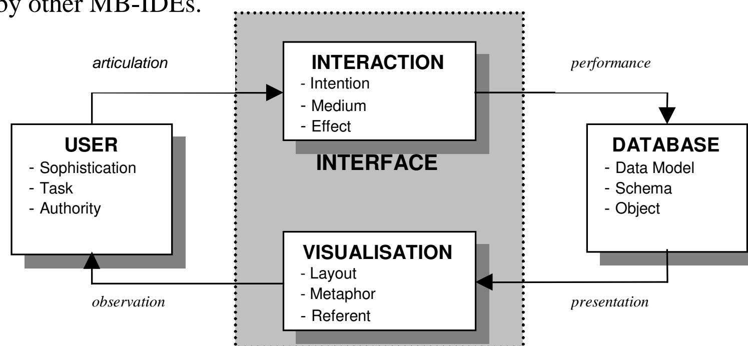


Figure14: DRIVE Framework for User Interfaces to Databases

DRIVE Meta-Model

The DRIVE system is centred on a meta-model of the IDS framework, and is captured in the meta-model as either classes, as shown in figure 15, or as properties, operations or triggers of those classes. There are 4 major meta-level classes for the framework: USER, INTERFACE, VISUALISATION and DATA. A given user interface to a database defined in DRIVE consists of instances of these meta-level classes. For example, the domain model for DRIVE is specified by defining a schema in the form of a set of DATA classes; similarly the presentation model is defined with a set of VISUALISATION classes. The meta-level classes act as templates for the classes which describe the interface to the database. DATA classes are free-form except that they must have property *has interfaces* which associate the data class with one or more interface class if the instances of the data in the database are to be available in the interface.

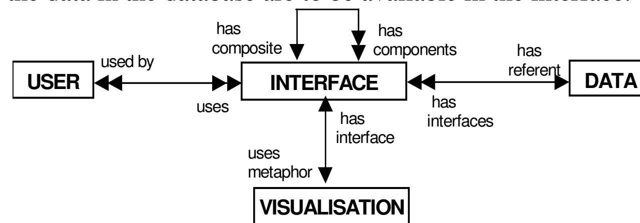


Figure 15: DRIVE Meta-model Showing Basic Template Classes for IDSs

While DRIVE employs a single modelling language, its model may be partitioned into a collaboration of several smaller dedicated models. The terms user model, domain model and interface model will therefore be used to refer to the corresponding sections of the IDS framework.

User Model

The user model is used to store the features of the users who will use this system. Many different users may be defined using the model, each with different levels of authority and sophistication. So far little use has been made of the user model other than user identification at start-up to present the appropriate interface. The model allows specialisation hierarchies of user or user groups to be constructed.

Domain Model

The domain model is basically the schema of the database. This describes the classes that are in use by the database as the enterprise model. These classes form the application to which the interface front-ends in more traditional MB-IDEs.

Interface Model

The DRIVE interface model concerns task and dialogue information and the mediation between presentation and domain information. Each interface artefact seen on the screen is represented by an AIO. The actual presentation behaviour is determined by an associated visualisation object (e.g. a form) which has standard presentation properties such as colour, position, and orientation. This information is traditionally stored in the presentation model in MB-IDEs. The interface classes

also model the response the interface should make given a particular set of inputs from the user and the database. This is referred to in the framework as "intention, medium and effect". These are modelled through the use of operations and triggers in the conceptual modelling language NOODL (Barclay, 1993). Operations are the intention and medium, triggers cause the effect. The effect may cause an event to take place in the database or the interface. The operations and triggers represent the form of the dialogue between the user and interface. It is worth noting here that all the DRIVE models are stored within the database itself, the database interface along with the database data. This has several useful side-effects (Mitchell, 1996b) including several forms of run-time customisation.

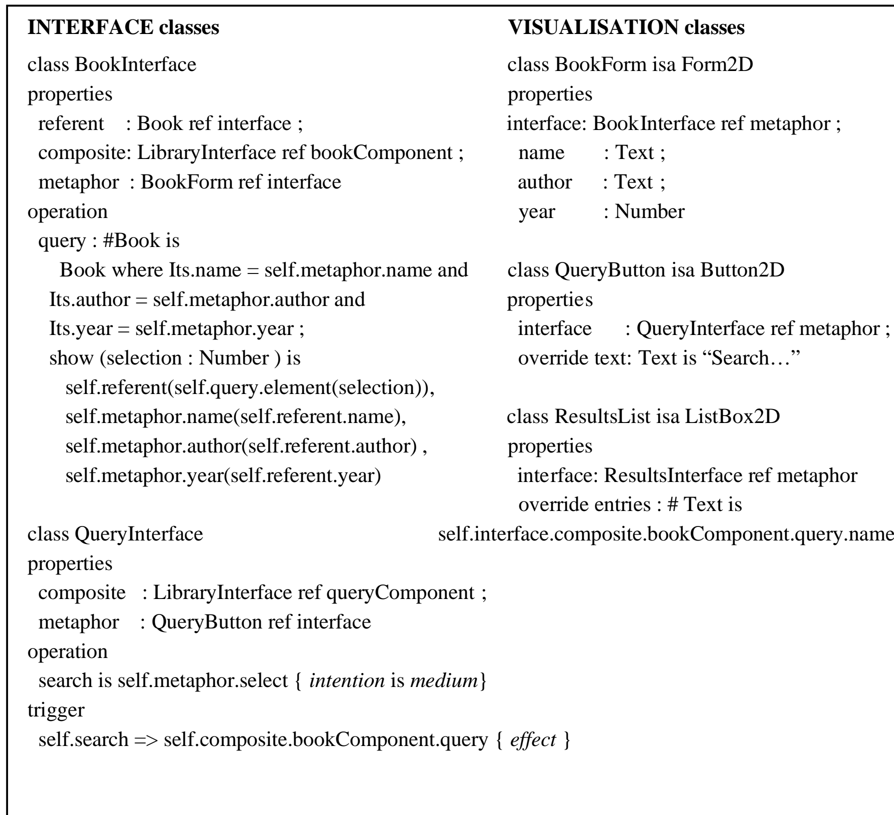
DRIVE Methodology

DRIVE was specifically designed to allow interfaces to object oriented databases to be developed. Given this, there is an implicit assumption that the first task involved after a general requirements analysis phase, which is unsupported by DRIVE, is to develop the domain model (or DATA classes of the framework). The NOODL schema for the Library database consists of a range of object classes, operations and constraints.

Having developed the Domain model for the database, the User model and Interface models can be described. The current User model in DRIVE is simple, allowing the interfaces available to a user and their sophistication and authority to be specified through the classes Interface, Sophistication and Authority respectively. These classes associated with the User class in the framework meta-model. In the case study there are two categories of USER class, Borrower and Librarian. These have been taken from the domain model and augmented with the following USER class properties.

```
accessors       : #BoolInterface ref users;
sophistication  : Sophistication ref user;
authority       : Authority ref user;
```

Given the Domain model and the User model, the Interface model may be defined. The IDS layout and visual appearance are interactively constructed by dragging and dropping from a dynamically registered palette of widgets. Each widget created corresponds to an instance of a visualisation class in the presentation model. These are then coupled to appropriate data and functionality through the properties, operations and triggers of interface classes. DRIVE uses model interpretation to permit the interface under development to be executed at any time, even when incomplete. As tasks are identified the specification of interface classes may be refined, whereby the user's intention, medium and effect may be specified. For example, in performing a database query, the intention is to search, the medium is the widget through which the task is performed and the effect is the query itself. Following the rules of the IDS template classes, this gives a NOODL interface class specification in terms of VISUALISATION and INTERFACE classes (some of which is shown in figure 16). The DRIVE methodology may be summarised as shown in figure 17.



INTERFACE classes

```
class BookInterface
properties
  referent    : Book ref interface ;
  composite: LibraryInterface ref bookComponent ;
  metaphor  : BookForm ref interface
operation
  query : #Book is
     Book where Its.name = self.metaphor.name and
    Its.author = self.metaphor.author and
    Its.year = self.metaphor.year ;
    show (selection : Number ) is
       self.referent(self.query.element(selection)),
       self.metaphor.name(self.referent.name),
       self.metaphor.author(self.referent.author) ,
       self.metaphor.year(self.referent.year)

class QueryInterface
properties
  composite   : LibraryInterface ref queryComponent ;
  metaphor    : QueryButton ref interface
operation
  search is self.metaphor.select { intention is medium}
trigger
  self.search => self.composite.bookComponent.query { effect }
```

VISUALISATION classes

```
class BookForm isa Form2D
properties
interface: BookInterface ref metaphor ;
  name       : Text ;
  author     : Text ;
  year       : Number

class QueryButton isa Button2D
properties
  interface      : QueryInterface ref metaphor ;
  override text: Text is "Search…"

class ResultsList isa ListBox2D
properties
  interface: ResultsInterface ref metaphor
  override entries : # Text is
self.interface.composite.bookComponent.query.name
```

Figure16: NOODL Interface Class Specification

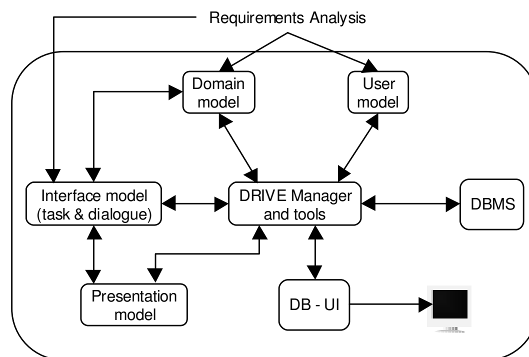


Figure 17: The DRIVE Interface Development Methodology

4 SUMMARY OF MODEL-BASED SYSTEMS

In this section the discussed MB-IDEs are evaluated across several dimensions, which are divided into sections examining models, architectures and tools. The section concludes with a discussion of the database-specific issues that this evaluation has raised. A question mark is used in table 4 to indicate that the referenced literature does not provide sufficient information to answer the section. Where necessary, additional references to abbreviations are explained in the text.

Models

This section examines the models and any associated modelling tools provided by the evaluated systems. These modelling tools provide a means of hiding the underlying (often complex) syntax of the modelling languages. Modelling tools include simple textual editors (Mastermind, DRIVE), forms-based tools, and graphical editors (HUMANOID, TADEUS, Adept).

- *Task model notation* – this model forms the basis of each of the evaluated systems. The task model utilises concepts found in the domain model, and is based on formalisms from HCI task modelling research. Notations include formal methodologies such as CSP (Hoare, 1985).
- *User model notation* – for the most part this is the most under-specified model. Formal notations are not utilised, and most systems do not describe how this model is utilised. Notations include rules that describe mappings and objects that capture facts about users that are exploited by mapping rules.
- *Domain model notation* – this model forms the basis of many MB-IDEs, providing information about the type of identified domain objects and their attributes and operations. Notations include informal references to domain objects; structured paradigms such as UML or ER diagrams; and the use of languages such as CORBA IDL or NOODL. In the specific case of databases, the domain model corresponds to the schema of the underlying application stored in a relational or object database. The concepts modelled in the domain model should provide the substance to the modelled tasks. At run-time, domain model concepts should provide the linkage between the underlying application and the runnable interface.
- *Dialogue model notation* – this model captures the sequencing information and complex interface semantics of the generated interface. Systems without an explicit dialogue model derive this information from their task or domain models. MB-IDEs which utilise a dialogue model use graphical notations such as Petri-nets (TADEUS) to capture the often complex intentions of the designer.
- *Presentation model* – this model allows the designer to specify abstractly the layout of the generated interface. The information captured by this model allows the designer to customise the generated interface. In some cases the presentation model can also allow the modelling of run-time application-dependent data, for instance to dynamically reconstruct the interface when application data structures change.

- *Modelling tools* – some systems provide tools that provide a convenient interface to the models under development. These tools include text editors, forms-based tools, and GUI editors.

	Dimension	Adept	HUMANOID	Mastermind	DRIVE	TADEUS
Models	*Task model notation*	CSP	No	CORBA IDL	NOODL ops	CSP
	User model notation	Rules	No	No	NOODL classes	Objects
	Domain model notation	Informal refs	Informal refs.	CORBA IDL	NOODL classes	UML
	Dialogue model notation	No	No	No	NOODL ops	Petri-nets
	Presentation model	No	Yes	CORBA IDL	NOODL classes	No
	- customisability	N/A	Yes	Yes	Yes	N/A
	- run-time data	N/A	No	Yes	Yes	N/A
	Modelling tools	GUI	GUI/Text editor	Text editor	GUI/Text editor	GUI
Architecture	*Methodology style*	Sequence (A)	Semi-sequential (A)	Ad-Hoc (S)	Ad-Hoc (S)	Sequence (A)
	- stages automated	Initial stage of each model → Interface	Models → Interface	Models → Interface	Model → Interface	Task → Dialogue → Interface
	Run-time system	Smalltalk	Interpreted	C++	Interpreted	Specification
	Propagation of changes	No	?	Yes	Partial	No
	Multi-platform support	P	?	P + E	No	P
	Generated interface style	WIMPS	WIMPS	WIMPS	F, 2D, 3D, DM	WIMPS+m
Tools	*Help generation*	No	Yes	No	No	No
	Undo support	No	Yes	No	No	No
	Design critics or advisors	No	?	No	Verifies against IDS	Simulation

Table 4: Summary of Characteristics of Evaluated Model-based Systems

Architecture

This section discusses how the architecture of the MB-IDEs affects the modelling capabilities of the system and the appearance and capabilities of the generated interface.

- *Methodology style* – some systems impose a rigid sequential structure on the order in which the models must be developed, with some systems allowing some of their models to be developed concurrently, and others favouring an ad-hoc development process. Furthermore, the methodology can either automate (A) some stages of the development process, or can be specification-based (S) – providing a complete specification of the proposed interface developed by the designer. While each evaluated system automates the production of the runnable interface from the model-based specifications, the specification-based systems also allow the task model to generate the initial state of their intermediate models.
- *Run-time system* – the runnable interface generated by the MB-IDE can either generate source-code in a programming language (e.g., C++ (Mastermind), or Smalltalk (Adept)), can produce a specification of the developed interface suitable for an existing UIDE or interface builder (e.g., TADEUS), or can produce a run-time interpretation of the interface (e.g., DRIVE).
- *Propagation of changes* – if a system allows the generated interface to be customised at run-time then the MB-IDE can attempt to propagate these changes to the underlying declarative models. If this facility is not available then any such modifications will be lost. Some systems (Mastermind) also allow changes made at design-time in one model to be propagated to the other models.
- *Multi-platform support* – the generated interface may be able to run on several platforms or operating systems utilising the platform's interface style (P). The MB-IDE may also generate different interfaces according to the envisaged operating environment (E), e.g. laptop, full-size screen, or kiosk.
- *Generated interface style* – this can be forms-based (F), direct manipulation (DM), WIMPS, 2D or 3D. In addition modal interaction with the generated interface can be supported (m).

Secondary tools

The declarative models provided by MB-IDEs produce a semantically rich source of interface and domain information. This information can be exploited by MB-IDEs to produce enhanced functionality in the following areas.

- *Help generation* – some systems automatically or semi-automatically generate part of the user help system of the generated interface.
- *Undo support* – some systems provide methods for automatically providing undo/redo facilities.
- *Design critics or advisors* – these tools can analyse the information contained in the various declarative models. They either verify that the design satisfies specified properties, simulate end-user interactivity, or produce statistics on the quality of the developed interface and its dialogue structure.

Database Issues

If MB-UIDEs are to provide a viable base for database UI generation, they must provide several database-specific facilities. These facilities may be captured in one or more of their models, and include:

- *Information passing* – both transient and persistent information needs to be used by (and within) an interface. A MB-UIDE should provide a means of describing this information flow, and methods for mapping the information to its equivalent domain concepts. This ability therefore requires the task or dialogue model to be capable of representing information flow, and to be able to map such information to concepts captured in the domain model.
- *Transactions* – the ability to be aware of database transaction has important implications for facilities such as undo. Transaction information should be accessible (possibly via a domain model) to allow potentially fine-grained transaction processing information to be captured through a task or dialogue model.
- *Database user modelling* – as previously discussed, the user model is the least exploited model. Databases have an in-built knowledge base of stored user information in terms of user authorities and access rights. If fully exploited, this information can produce interfaces which closely reflects a user's mental model of the information available or applicable to them.
- *Database-specific visualisations* – the discussed systems utilise a limited (and frequently fixed) set of visualisations, with the developer typically constructing an interface from a set of simple CIO building blocks. If MB-UIDEs are to produce useful database interfaces, they must be capable of providing an open architecture which will allow developers to use existing data visualisation and schema representation widgets and component libraries.

5 CONCLUSIONS

MB-IDEs can provide a novel means of generating user interfaces to represent the domain knowledge captured in a database. While the sophistication of the generated interface depends largely on the capabilities of the dialogue and presentation models, the underlying functionality of the interface is primarily a reflection of the richness of the domain and task models. When using a MB-IDE to produce an interface to a database, the primary concern of the interface designer is therefore the creation of the domain and task models, and the subsequent linkage of domain model concepts to the underlying database schema and the services provided by the DBMS.

With the exception of Adept and HUMANOID, the evaluated MB-IDEs use an object-oriented realisation of the domain model. While this provides an easy mapping from an OODBMS schema to the domain model, a relational schema would require additional transformations into its object-oriented equivalent. The question of utilising Mastermind's CORBA-based model repository can be reduced to producing a wrapper which will export a call interface suitable for use with a database API. While both HUMANOID and Adept utilise domain concepts in the

construction of the user interface, the domain model is not explicitly developed by the designer. In the specific case of interfaces to databases, the lack of an explicit domain model results both in a loss of domain knowledge and difficulties in maintaining the links between domain concepts and the generated interface.

The task-orientated approach to developing interfaces adopted by the evaluated MB-IDEs is based upon a sound platform of long-standing and well understood HCI research, utilising the knowledge captured during the early requirements analysis phase of developing a database application. In addition, the use of declarative models focuses the developer's attentions on what the interface should represent and do rather than how this should be realised.

The evaluated MB-IDEs take different stances concerning methodological style. The sequential approach of the automated design tools (Adept, TADEUS, and to a lesser degree HUMANOID) will be seen by advocates of structured methodologies to be systematic tools, whereas the more ad-hoc approach favoured by the specification-based tools (Mastermind and DRIVE) provides the interface developer with arguably more control over the finished interface.

Whilst most of the evaluated MB-IDEs propose a user model, the facets of users or user groups which can be captured by this model remain unclear. Furthermore, the manner in which the user model is utilised is frequently under-specified. In the specific case of generating interfaces to databases, the potential for utilising the user model to capture features such as the access rights and requested functionality inherent in the DBMS data dictionary remains as yet un-exploited.

This paper has found that although the referenced MB-IDEs utilise some form of a domain model (or domain concepts), the method by which the domain model is linked to an underlying database application remains unclear. If MB-IDEs are to prove to be a useful vehicle for developing user interfaces to databases they must cater for database-specific concepts such as query construction and execution, rollback, schema modelling and visualisation, and the presentation of non-forms-based graphics. It also remains unclear from the referenced literature how fundamental tasks such as the construction of database queries from user information gathered through the completion of query sub-tasks is to be achieved; indeed there is still a need for specifying how query results are to be returned to the user. The capabilities of the domain model could also be extended to utilise the meta knowledge contained in database-specific formalisms such as ODMG or ER diagrams to allow, for example, the integrity of the database to be maintained through the constraints inherent in such formalisms.

Acknowledgements

We are indebted to the contribution made to this work by Kenny Mitchell for his discussions on the DRIVE presentation model. This work is funded by UK's Engineering and Physical Sciences Research Council (EPSRC).

6 REFERENCES

Balzert, H., et al. (1996) The Janus Application Development Environment: Generating More than the User Interface, in *Computer-Aided Design of User Interfaces* (ed. J. Vanderdonckt), Namur University Press, Namur, 183–205.

Barclay, P.J. (1993) Object Oriented Modelling of Complex Data with Automatic Generation of a Persistent Representation. PhD Thesis, Napier University, Edinburgh.

Bodart, F., et al. (1995) Towards a Systematic Building of Software Architectures: The TRIDENT Methodological Guide, in *Interactive Systems: Design, Specification and Verification*. Berlin: Springer, 77–94.

Cattel, R., et al. (1996) The Object Database Standard: ODMG 2.0. Morgan Kaufmann.

Devin, L. (1997) Delphi 3 Technology Overview, http://netserv.borland.com/delphi/papers/techover.

Elwert, T., Schlungbaum, T. (1995) Modelling and Generation of Graphical User Interfaces in the TADEUS Approach, in *Design Specification, and Verification of Interactive Systems* (eds. P. Palanque and R. Bastide). Wien, Springer, 193–208.

Foley, J. Sukaviriya, P. (1995) History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation, in *Interactive Systems: Design, Specification and Verification*. Berlin: Springer, 3–14.

Hoare, C.A.R. (1985) Communicating Sequential Processes. Prentice-Hall.

ISA (1995) The ISA Dialog Manager: http://www.isa.de/en/idm/

Johnson, P., et al. (1991) Task related knowledge structures: analysis modelling and application, in *People and Computers IV: From Research to Implementation*, (eds. D. Jones and R. Winder). Cambridge University Press, Cambridge.

Johnson, P., Johnson, H., Wilson, S. (1995) Rapid Prototyping of User Interfaces Driven by Task Models, in *Scenario-Based Design*, (ed. J. Carroll). John Wiley & Son (London), 209–246.

Kelly, C. & Colgan, L. (1992) User Modelling and User Interface Design, in *People and Computers VII*, (eds. A. Monk, D. Daiper, M. Harrison). Cambridge University Press, Cambridge, 227–239.

Kennedy, J.B. & Barcley P.J. (eds.) (1996) Interfaces to Databases (IDS-3), in *Proceedings of the 3rd International Workshop on Interfaces to Databases*, Napier University, Edinburgh, 8–10 July 1996. Electronic Workshops in Computing, Springer.

Luo, P., Szekely, P., & Neches, R., (1993), Management of Interface Design in HUMANOID, in *Proceedings of InterCHI'93*, Amsterdam.

Mitchell, K., Kennedy, J., Barclay, P. (1995) Using a Conceptual Data Language to Describe a Database and its Interface, in *British National Conference on Databases 13*, Manchester, England, 101–119.

Mitchell, K., Kennedy, J., Barclay, P. (1996a) A Framework for User Interfaces to Databases, in *ACM International Workshop on Advanced Visual Interfaces*. Gubbio, Italy.

Mitchell, K., Kennedy, J. (1996b) DRIVE: An Environment for the Organised Construction of User-Interfaces to Databases, in *International Workshop on Interfaces to Databases 3*. Edinburgh, Scotland.

Mitchell, K. (1997) Three Dimensional Database Environments. PhD Thesis, Napier University.

Myers, B.A. (1995) User Interface Software Tools, in *ACM Transactions on Computer-Human Interaction*, **2(1)**, 64–103.

OMG (1995) CORBA: Architecture and Specification. Object Management Group Publication Services.

Oracle (1996) Oracle Forms Developer's Guide, Release 4.5, Part No. A32505-2.

Puerta, A. (1996) The Mecano Project: Comprehensive and Integrated Support for Model-based Interface Development, in *Computer-Aided Design of User Interfaces* (ed. J. Vanderdonckt). Namur University Press, Namur, 19–36.

Rumbaugh, J. et al. (1991) Object-Oriented Modelling and Design. Prentice-Hall.

Schlungbaum, E. (1996) Model-based User Interface Software Tools - Current state of declarative models. Graphics, Visualization and Usability Centre, Georgia Institute of Technology, GVU Tech Report #96-30.

Schlungbaum, E., Elwert, T. (1996a) Dialogue Graphs - A Formal and Visual Specification Technique for Dialogue Modelling, in *Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*, Sheffield Hallam University (eds. C.R. Roast and J.I. Siddiqi). Springer-Verlag London.

Spaccapietra, S. & Jain, R. (eds.) (1995): Visual Database Systems 3, Visual Information Management, in *Proceedings of the 3rd IFIP 2.6 working conference on visual database systems*. Chapman & Hall.

Szekely, P. (1994) User Interface Prototyping: Tools and Techniques. Technical report, Intelligent Systems Division, University of Southern California.

Szekely, P., et al. (1996) Declarative Interface Models For User Interface Construction Tools: The MASTERMIND Approach, in *Engineering For Human-Computer Interaction*.

Wierse, A., et al. (eds.) (1995) 2nd Workshop on Database Issues for Data Visualization, Atlanta, Georgia,. - Database issues for data visualization: IEEE Visualization '95 Workshop. Springer, London.