# Data Management: From EXPRESS Schemata to User Interface

Alain Plantec, Vincent Ribaud

# Data Management: From EXPRESS Schemata To User Interface [1]

A.Plantec and V.Ribaud
LIBr & Syseca

LIBr: Faculté des Sciences, BP 809,
29285 Brest Cedex, France
Syseca: 34 quai de la Douane, 29200 Brest, France
E-mail: {plantec,ribaud}@univ-brest.fr

**Abstract**

User Interface Management Systems (UIMS) allow interface designers to create a complete and working user interface. UIMSs considered in this article are those based on object oriented technology. Inside the UIMS, the interface is described in a distinctive interface specification language. This specification is then translated into programming toolkit functions.

STEP is an ISO standard (ISO-10303) for the computer-interpretable representation and exchange of product data. Within STEP, *EXPRESS* language is meant to describe object-oriented data models called schemata.

For a data management application, user interface can be generated from data schemata. A mapping between EXPRESS language constructions and interactive objects classes can be established. This states the representation rules of an *EXPRESS* schema. *User interface specification* can be inferred from *EXPRESS* schemata by using the *building rules* for graphic objects and for objects behaviour. The generation is partly automatic, partly controlled by the designer.

The alterations of the *building rules* enables the designer to work at the meta-generation level. This meta-generation is supported by a structured description of the building rules.

The capabilities for meta-generation allow the designer to adapt the generation. The main goal remains the generation of an abstract interface specification. This specification will then be translated into the distinctive toolkit target. Both the generated interface and the building process have reflective aspects.

[1] Final version submitted to Journal of Computing and Information, Vol. 2, No. 1, 1996

# 1 Introduction

Data management uses a lot of formal models (entity-relationship, relational, object-oriented...) at different phases of system life cycle. The programming environments often offer the possibility to describe all the used models. This description is structured in a dictionary or meta-model, which, following Codd's idea [4] is usually implemented with the same programming environment constructions as the models themselves. Software engineering tools and programmers use these meta-informations to transform models from one phase to another or to generate programs and documentation.

*STEP* is an ISO-10303 standard for the computer-interpretable representation and exchange of product data [10]. The data models, called schemata, are described in the *EXPRESS* language [11]. The data manipulation functions in a given programming language are drawn automatically from these schemata. This automatically-obtained component is called an *SDAI* (Standard Data Access Interface) [8].

Parallel with the idea of data independence used in database-management (and in the *SDAI*), human interface construction tools and methods benefit from the notion of user interface independence, unbinding the user interface design from the underlying implementation [5].

The basic idea in this article is that, for a data management application, the user interface can be generated from the data schemata. The generated user interface and the generated SDAI can be put together to give a stand-alone data management application. This application aims to insert, delete, update and query data in a database.

User interface is built in three main steps: transformation, elaboration and translation. These three steps act on an intermediate representation of the user interface, modeled in *EXPRESS* schemata. This intermediate representation is named *User Interface Specification*.

Initially, the transformation step automatically builds a first version of the specification, drawn from the conceptual schema of application data.

During the elaboration step, the interface designer is able to update the specification. Finally, the translation step automatically produces a working interface which depends on the execution environment of the application.

An interesting perspective of this building process is that both the generated interface and the building process itself have reflective aspects.

This article is organized as follows. In Chapter 2, we detail the *STEP* point of view. In Chapter 3, we present User Interface Management Systems (*UIMSs*). In Chapter 4, we describe the data management application building process. In Chapter 5 we describe the *user interface specification*. In Chapter 6 we talk about reflection. In Chapter 7 we give the designer interactions in greater detail. We finish with related work, perspectives and conclusion.

# 2 STEP

*STEP* (ISO 10303) is an international standard for the computer interpretable representation and exchange of product data [10]. This standard provides a modelling language, *EXPRESS* (ISO 10303-11) [11], a neutral data encoding (ISO 10303-21) [12] and an application program interface to data called Standard Data Access Interface (*SDAI*, ISO 10303-22) [8]. The NIST (National Institute of Standard and Technology) pioneered work in this field and promotes public domain tools [17, 13].

The data are kept in a storage facility called a repository. *STEP* considers three kinds of repositories: (1) the memory, (2) ASCII files in which data are encoded in the *STEP* neutral exchange format, and (3) a database.
From a traditional point of view in the field of database management, *EXPRESS* can be considered as a Data Definition Language (DDL) and the *SDAI* as the Data Manipulation Language (DML).

## 2.1 The EXPRESS language

*EXPRESS* is an object-oriented modelling language. The application data are described in schemata and a schema can reference other schemata. This allows the designer to write generic schemata referenced by more specific ones.

A schema owns the types definitions and the objects descriptions of the application called *Entities*. An entity is made of attributes and constraints descriptions and can inherit from others entities. The constraints expressed in an entity definition can be of several kinds [11], briefly:

- the *UNIQUE* constraint allows entity attributes to be constrained to be unique either singly or jointly (e.g any one value of that (these) attribute(s) is (are) associated with only one instance of the owner entity),

- the *DERIVE* constraint is used to represente computed attributes. Such constraint specifies the way derived attributes are computed,
- the *WHERE* clause of an entity constraints each instance of an entity individually,
- the *INVERSE* clause is used to specify the inverse cardinality constraints.

*EXPRESS* allows the definition of global rules. These rules are used when either all instances of a given entity or instances of at least two entities need to be examined concurrently to determine whether a given constraint is satisfied. An example of *EXPRESS* schema is shown in figure 1.

```
schema BarAndBeverages;

   type BeerColor = enumeration of (lager, stout);
   end_type;

   entity Bar;
         name : string
         town : string;
         beverages : list [1:?] of Beverage;
   end_entity;

   entity Beverage abstract supertype of (Wine, Beer);
         name : string;
         percentage : real;
      inverse
         bars : set [0:?] of Bar;
      where
         wr1 : percentage >= 0;
   end_entity;

   entity Wine subtype of (Beverage);
         vintage : string;
         year : integer;
      where
         wr2 : Beverage\percentage <= 14.0;
   end_entity;

   entity Beer subtype of (Beverage);
         color : BeerColor;
      derive
         maxCans : real:= computeMaxBeerCans(percentage);
   end_entity;

end_schema;
```

Figure 1: A simple EXPRESS schema

## 2.2 The SDAI

The *SDAI* defines an access protocol for *EXPRESS*-defined databases and is defined independently from any particular system and language.

The representation of this functional interface in a particular programming language is referred to as a language binding in the standard. As an example, ISO 10303-23 is the *STEP* part describing the C++ *SDAI* binding [9].

The existence of the *SDAI* enables each vendor to write a repository interface that corresponds to a common standard. In turn, an application developer can then insure portability of his application among all systems supported by these vendors.

Usually, a *STEP* implementation platform includes not only an *SDAI* binding in a particular programming language and for particular repositories but also a database physical schema builder, parsing *EXPRESS* schemata and producing DDL and DML programs for a specific repository. The main point is that those specific repositories features are not and need not be known by the *SDAI* user.

The main goals of the *SDAI* are [8]:

- to access and manipulate data which are described using *EXPRESS* language so that access to a database happens through a conceptual schema, not a physical schema,
- to allow access to multiple data repositories by a single application at the same time,
- to allow commit and rollback on a set of *SDAI* operations,
- to allow access to the *EXPRESS* definition of all data elements that can be manipulated by an application process, and
- to allow the validation of the constraints defined in *EXPRESS*.

# 3 UIMS

User interface is built on the libraries of the operating system or on windows system like Windows, X- Window, etc. Programming toolkits attempt to hide the complexity of programming these systems by providing especially designed routines that handle standard widgets such as *windows*, *scroll-bars*, *menus*, *data-entry fields*, *buttons*, and *dialog boxes* [25].

The family of toolkits we are considering in this article, organizes interface objects in a hierarchy of classes, in an object-oriented way. (In this article, the name *toolkit* is used to refer to these graphics interface toolkits).

In [18], D.A.Norman and S.W.Drapper talk about the use of a User Interface Management Systems (*UIMS*) and its functionalities:

> *An UIMS allows a user interface designer to specify his interface in a high-level language. The UIMS transforms these specifications into interface objects, the management of these objects and the management of the dialogue with the application.*

As a general rule, the designer does not program in this high-level language, but uses an interactive interface construction tool.

In this article, the high-level language is made of *EXPRESS* schemata and their instances. We think that our ideas apply to different object-oriented toolkits. The validation of some ideas required the implementation of software components, first with the *tcl/tk* environment [19], currently with the *ILOG Views* library [7].

# 4 The data management application building process

The purpose of this section is to show how a complete and working data management application can be built from data model specifications. The functionalities of the generated application are those offered by the *SDAI*, mainly update and query data hold in a repository. The key components of this process are a *SDAI* and a *User Interface Specification* directly produced from the *EXPRESS application data schema.*

## 4.1 A *Compilation* problem

The problem can be seen as belonging to the field of compilation: an intermediate representation is built through an *EXPRESS* schema analysis process and this representation is used by a synthesis process to produce a data management program target for a particular *toolkit* or language.

The user interface *building rules* are then defined in the synthesis program itself. The wellknown feature of this approach is that only the synthesis program has to be rewritten in the case of another *toolkit* target.

We feel that the *toolkits* share many common features and are organised in a similar manner. So, it is possible to describe the user interface in a common specification language and translate this specification into a given *toolkit.*

Then, the compilation process can be re-specified (figure 2). The data structure intended to store any *EXPRESS* schema representation is usually called the *EXPRESS dictionary* (meta-schema level). In the same manner, the data structure intended to store a *user interface specification* is named the *user interface dictionary* (meta-schema level).

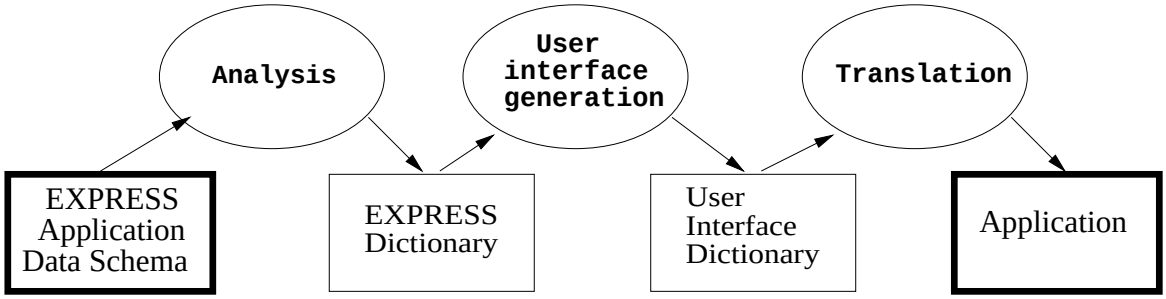


Figure 2: The compilation point of view

The *user interface dictionary* is defined by a set of *EXPRESS* schemata referred to as the *user interface dictionary schema*. Assuming that the specification language is well designed for a given *toolkit*, translating *user interface specification* into this *toolkit* is a simple process.

## 4.2 The building process

Figure 3 shows the entire building process. This process is made of four sub-processes. The *Analysis* process reads the *EXPRESS application data schema* and fills an *EXPRESS dictionary*. This dictionary is used by the *Transformation* process which builds the *user interface specification*. The *Translation* process translates from the *user interface specification* to a *user interface program* and a *SDAI* is constructed directly from the *EXPRESS application data schema*. Finally, the *user interface program*, the *toolkit* and the *SDAI* libraries are linked together to build the *data management application*.

### 4.2.1 The analysis sub-process

This process reads the *EXPRESS application data schema* and builds an internal representation. This internal representation is made of instances of the *EXPRESS dictionary*. This dictionary stores information about the object-oriented data structures and about the constraints expressed in the model, especially about the collections cardinalities, the unicity constraint of attributes, the relationships between entities and the derived attributes computation.

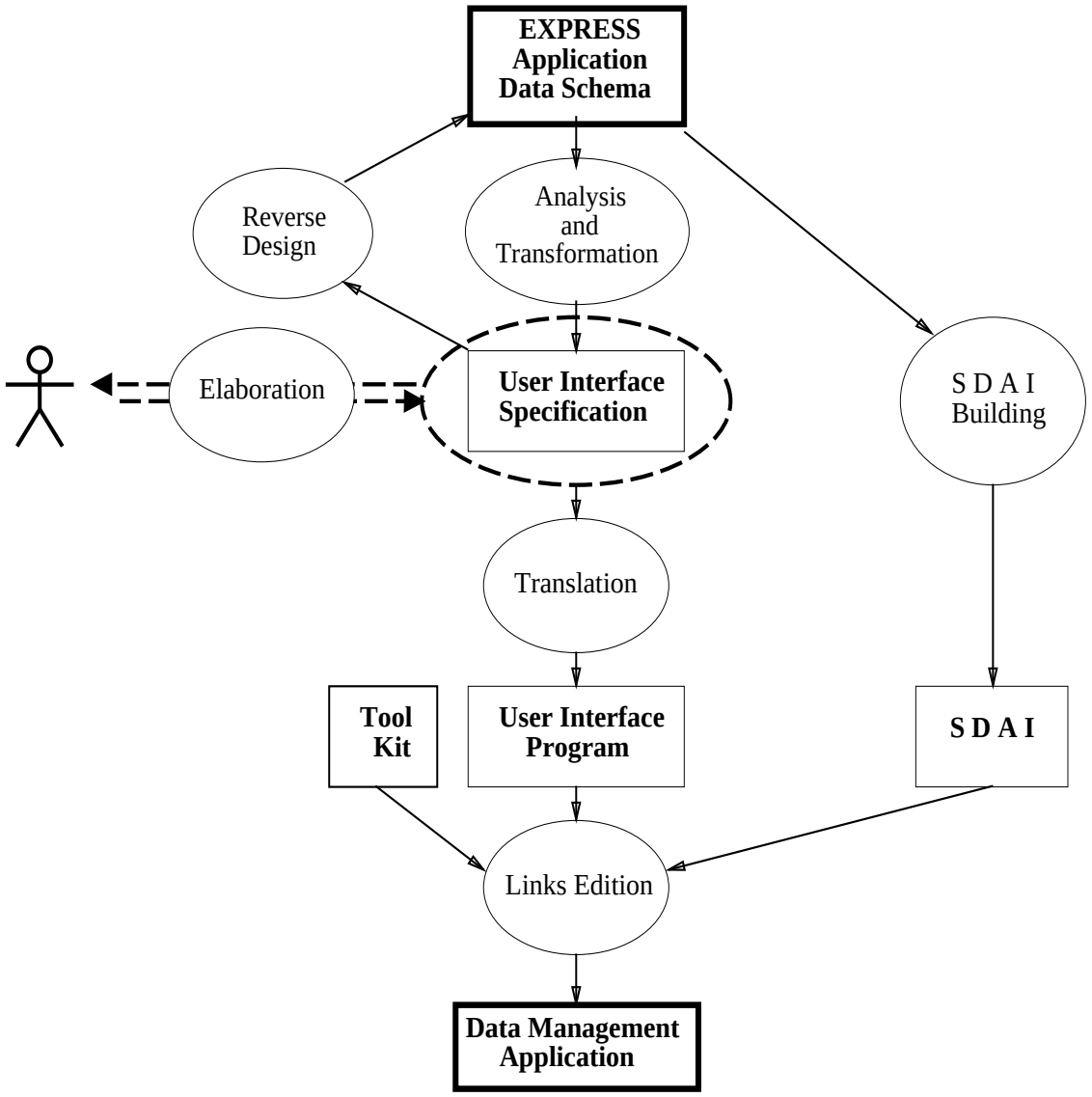


Figure 3: The Data Management Application Building Process

### 4.2.2 The transformation sub-process

This process takes place after the analysis process and uses the *EXPRESS dictionary* to build a *user interface specification*. There are eight kinds of *building rules*.

1. *Correlation rules*. These rules map *EXPRESS* built-in types to the *toolkit* basic objects and to type dependent behaviour. As an example, the *INTEGER*, *REAL* and *STRING EXPRESS* types are represented by an *entry* object and a procedure which is intended to check the input.

2. *Context dependent correlation rules*. They allow multiple presentation for *EXPRESS* types. The choices are context driven. For example, the representation of an entity reference could be the complete presentation of the referenced entity (we call it direct representation), a *button* or *menu* item (indirect representation) allowing access to the direct representation. Another example is the collection *building rules*. They describe the way collections are represented. The representation of a collection depends on the base element type. A collection of simple element type is represented by a vector of direct representations. A collection of collections can be represented by a matrix. The basic element of the matrix can be an *entry* (collection of collections of simple type), or a *button* (collection of collections of entity reference or of another collection) allowing access to the base collection base type representation.

3. EXPRESS *constructors rules*. They map *EXPRESS* constructors (*ENTITY, SELECT, ENUMERATION*) to their representation. For example, an *ENTITY* is represented by a frame containing the attributes representation, an *ENUMERATION* or a *SELECT* type by a listbox.

4. *Inheritance rules*. They describe the representation of the inherited entities attributes in an entity representation. If an entity named A inherits from another named B, the representation of A is the union of the B and A representation.

5. *Relationships rules*. They describe the representation of the *EXPRESS INVERSE* clause. The *INVERSE* clause of an entity is represented by a *button* which is intended to call a list representation of the inversed attribute.

6. *Constraints rules*. They map *EXPRESS* expressions to procedures. These procedures are called by a *button* (in case of a *EXPRESS WHERE* clause) or for computation of the derived attributes values.

7. *Default behaviour rules*. They state the link between the user interface and the *SDAI*. The default behaviours are represented by *buttons* or *menu items* calling

the *SDAI* functions. There are four kinds of default behaviour: (1) the reading and the writing of data, (2) the commit/rollback functionalities, (3) the query functionality and (4) the constraints validation.

8. *Default schema navigation rules.* They allow the representation of a root *menu* in a root window. The default root *menu* consists in items for the calling of the representation of the schema entities, for the calling of the referenced schemata navigation root window, for the execution of the schema global rules.

### 4.2.3 The designer's elaboration

The *Analysis* and *Transformation* processes transform the *EXPRESS application data schema* into a first version of the *user interface specification*. Within the process of interacting on the *user interface specification*, the interface designer elaborates a more and more accurate version until a satisfying solution is obtained.
Designer interactions are detailed in section 7.

### 4.2.4 The translation sub-process

This process consists in the translation between the *user interface specification* and the *user interface program*. The translation rules are *toolkit* dependent. This is a direct translation from objects of the *user interface specification* to the *toolkit* objects since representations choices have already been made during the *Transformation* process and by designer's elaboration.

### 4.2.5 The reverse design sub-process

Since all components of an *EXPRESS* schema are stored in the *EXPRESS dictionary* [8], it is possible to rebuild the *EXPRESS application data schema*. Because of the possible structural changes made by designer's elaboration, the original schema must be regenerated in order to build the *SDAI* or to use the schema for documentation.

The feature of reverse design appears as an implementation choice because another solution would be to rebuild the *SDAI* directly from the *user interface specification* which owns the *EXPRESS dictionary*. But this second solution excludes the use of the tools now available.

# 5 The user interface specification

The application of the building rules produces the description of interface objects: the *User Interface Specification*. The structure of these description is specified with

*EXPRESS* schemata. The *user interface specification* is then a set of instances of the *user interface specification schemata.*
These schemata are:

- the existing *SDAIdictionary* schema [8], which describes the application data,
- the *CoreWidget* schema, which describes the *toolkit* objects,
- and three different schemata for user interface objects:
  1. the *SDAIdictionaryPresentation* schema describes how the *EXPRESS* schema structures are represented.
  2. the *SDAIdictionaryAccess* schema allows *SDAIdictionary* entity references from the *SDAIdictionaryPresentation* schema, and
  3. the *SDAIentityBehaviour* schema contains the names of functions called for the default behaviour of an entity representation.

The *user interface specification schemata* and their relationships are shown in figure 4

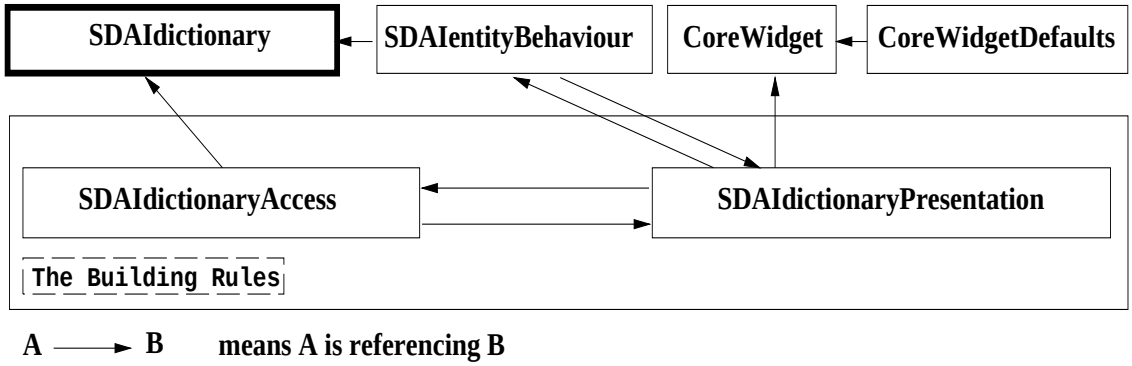


Figure 4: The user interface specification schemata

## 5.1 SDAIdictionary

The *SDAIdictionary* is the *EXPRESS dictionary schema.* It defines the *SDAI* data dictionary which stores information about schemata describing instances operated on by the *SDAI* [8]. Because the schemata *SDAI* operates on are defined in *EXPRESS*, the structure of this dictionary schema reflects the structure of *EXPRESS* itself.

Figure 5 shows the *EXPRESS* entity *entity_definition* describing an *EXPRESS* entity type.

```
entity named_type abstract supertype of (oneof (entity_definition, defined_type));
      name : express_id;
      where_rules : list [0:?] of where_rule;
end_entity;

entity entity_definition subtype of (named_type);
      supertypes : list [0:?] of unique entity_definition;
      attributes : list [0:?] of unique attribute;
      uniqueness_rules : set [0:?] of uniqueness_rule;
      complex : boolean;
      instantiable : boolean;
      independent : boolean;
      parent_schema : schema_definition;
end_entity;
```

Figure 5: The EXPRESS *entity_definition* entity

## 5.2 CoreWidget

The *CoreWidget* schema aims to describe all *toolkit* base widgets. It should be as general as possible and should not reflect the structure of a specific toolkit.

The schema is mainly made of the abstract entity *Widget* and all of its subclasses: *Label*, *Entry*, *Text*, *Button*, *Menu*, *Listbox*, etc. Figure 6 shows the *Widget* and the *Button* entities.

```
entity Widget abstract supertype of (oneof (Menu, Text, Button, Entry, Label,
...));
      class : string;
      backgroundColor : string; ...
   unique
      u1 : class;
end_entity;

entity Button supertype of (oneof (CheckButton, RadioButton, MenuButton))
  subtype of (Widget);
      height : optional Size;
      sensitive : boolean;
      clientDataName : string;
end_entity;
```

Figure 6: The *Widget* and *Button* entities

For each of these entities a *default values* entity is defined which owns the attributes default values. If *A_Default* is the name of the *default values* entity of *A*, then *A_Default* inherits from *A* and sets all the mandatory attributes to their default value in a *WHERE* clause. Those particular entities are defined in the *CoreWidgetDefaults* schema. The inheritance use for representation of particular values has

already been specified by Kramer, Morris and Sauder in [13].

## 5.3 The building rules schemata

```
entity  entity_definition_access;
        accessTo : entity_definition;
    inverse
        presentation : set  [0:?] of  entity_definition_presentation for  subject;
end_entity;

entity  entity_definition_presentation subtype  of  (named_type_presentation);
        subject : entity_definition_access;
        globalframe : Frame;
        attributes_presentation : list  [0:?] of  attribute_presentation;
        behaviour : entity_definition_behaviour; ...
end_entity;
```

Figure 7: The presentation rules of an *EXPRESS* entity

The *SDAIdictionaryPresentation* defines how *EXPRESS application data schema* structures are represented. It aims to connect the *SDAIdictionary* and the *CoreWidget* entities.

For each *SDAIdictionary* type, entity or rule definition, an *SDAIdictionaryPresentation presentation definition* entity which describes its representation is defined.

The types of attributes of *presentation definition* entities consist in *SDAIdictionaryPresentation* types, *SDAIdictionaryAccess* types or *CoreWidget* types.

The *SDAIdictionaryPresentation* schema indirectly references the *SDAIdictionary* schema via the *SDAIdictionaryAccess* schema. The *SDAIdictionaryPresentation* and *SDAIdictionaryAccess* schemata specify the static aspects of the *building rules*. They are called the *building rules schemata*.

As an example, the entity presentation rules are partly reproduced in figure 7.

## 5.4 SDAIentityBehaviour

The *SDAIentityBehaviour* schema describes a part of the entity presentation behaviour. This schema defines one entity: the *entity_definition_behaviour* entity shown in figure 8. These entity attributes are the names of the SDAI procedure for creating, updating or deleting an entity. These names are defined in the derived clause since they can be computed from both the *entity_definition* and the *entity_definition_presentation* attributes.

```
entity  entity_definition_behaviour;
        subject : entity_definition;
    derive
        create : string:= ComputeEntityCreatingProcName(subject, presentation);
        update : string:= ComputeEntityUpdatingProcName(subject, presentation);
        delete : string:= ComputeEntityDeletingProcName(subject, presentation);
    inverse
        presentation : set  [1:1] of  entity_definition_presentation for  behaviour;
end_entity;
```

Figure 8: The *entity_definition_behaviour* entity

# 6 Reflection

A reflective system is a system which incorporates structures representing (aspects of) itself. The sum of these structures is called the self-representation of the system. The system and its self-representation are causally connected: if one of them changes, this leads to a corresponding effect on the other [15].

## 6.1 Reflection of the generated interface

The set of schemata described in section 5 are meta-schemata. They contain meta-objects which hold knowledge about the objects to generate, the interface objects. According to Patti Maes's definition in [16], a meta-object holds, in an exhaustive manner, information concerning the object implantation, inheritance, instanciation, behaviour, etc.

Inside the building process, the meta-schemata and their meta-objects are the self-representation of the generated user interface. The final aim of this building process is to get a stand-alone interface, separated from the building process and hence of its self-representation. We can say that the generated application is not reflective.

The translation process should be either a compiler or an interpreter. The interpreter allows the designer to measure the effects of the specification while elaborating the interface. This is defined in [3] as a *user interface specification environment* rather than a UIMS.

In the building process, the self-representation is stored in *EXPRESS* schemata and then manipulated through a *SDAI*. The final generated application is a data management application, and therefore owns a *SDAI* in order to manipulate its own data. Incorporating the meta-schemata and an interpreter into the final generated

application makes the application reflective. Because it owns a self-representation (the meta-schemata) and the causal connection (the interpreter), the application is able to alter its own user interface.
The perspectives of this reflection are of two types:

- The application alone is allowed to alter itself. The application tracks the user's actions and tries to adapt to improve its performances: by creating new *menus* or keyboard short-cuts following frequently used paths, by proposing defaults values after detecting repeated value entry in a given field, etc.

- The user itself is allowed to act on the system. Using the same (or limited) functionalities of the building process, the user is able to act upon the interface. He can specify an alteration, which the application will carry out by altering his self-representation.

## 6.2 Reflection of the building process

We consider now instances of meta-schemata no longer as meta-objects about user interface objects but as building process objects. The production of these objects is controlled by rules. These rules are described in section 4.2.2.

Most of these rules are structured in *EXPRESS* schemata. This allows us to store a representation of these rules in a dictionary. This rules dictionary is a structure within the building process which describes the knowledge that the building process has about itself, and is a self-representation of the building process. The building process is then reflective and is able to alter itself: we called this feature meta-generation.

# 7 The designer interactions

As shown in figure 3, the designer can interact on the *user interface specification*. Designer interactions fit into three categories of operations:

1. Resources edition. The presentation of the interface is updatable through modifications of graphic attributes such as color, size, font, etc,

2. Customization. Composition of objects can be modified, as well as the access path between interface components,

3. Migration. Major changes of the objects structure are allowed, such as adding an attribute or modifying the hierarchy path.

## 7.1 Evolution

Once an initial *user interface specification* is obtained by transformation of the *EXPRESS application data schema*, the designer elaborates this specification by making successive changes. We then use the two major approaches of object oriented design: *transformation* and *elaboration*.

The *transformation* approach makes an explicit distinction between conceptual level (data schema) and logical level (interface specification) and advocates a transformation process between levels [24]. The *elaboration* approach is based on a unique level and advocates a process by successive refinements [21, 2].

A problem arises when a change is needed in an *EXPRESS application data schema* after successive elaborations of the *user interface specification*.
The existence of two independent levels makes its difficult to go back from the interface level to the data schema level, and it is sometimes impossible to maintain consistency between the two levels. Regenerating the interface from the updated *EXPRESS application data schema* while keeping the user's transformations would take a sophisticated mechanism to apply all the user's elaborations to the initially retransformed interface.
The fashion in which data schema changes are done solve the problem: these changes are made both in the *user interface specification* and in the *EXPRESS application data schema*.

*EXPRESS* schemata migration is discussed in [22, 23]. D.Sanderson follows the point of view of object-oriented data schema evolution exposed in [1, 20, 14]. This approach is based on invariants and on operations. Operations allow schema modifications, and invariants must be preserved across alterations.

For each data schema evolution operation, changes to be made in the corresponding user interface are defined. The building process owns a description of the *EXPRESS application data schema*, the *SDAIdictionary*. Evolution operations are applied to this description, while corresponding changes are applied to the *user interface specification*. In fact, we could say that in order to elaborate a structural change on the interface, we elaborate a change on the *EXPRESS application data schema* and we apply a local transformation to get the new presentation.

## 7.2 Consistency

User interface under construction is stored as instances of several schemata. The meaning of the interface is controlled by the constraints of the schemata. These constraints are meant to be well-designed so that any meaningless interface cannot

happen. Assuming the schemata design is correct, we are able to judge the effects of these operations according the following simple principle which guarantees interface consistency:

> Any effects which respect the constraints are valid and the corresponding operation is allowed.

The functionalities of the application are those provided by the SDAI. Interface callbacks call on SDAI functions. Consistency between the interface and the SDAI should be guaranteed too. This is discussed in the next section.

## 7.3 Resources edition

Resources editing consists in the modification of the basic parameters of the presentation. These functions are standard in an *UIMS*. They are similar to the functions available in an interactive resource editor such *ResEdit* on the Macintosh or a resource description language such *User Interface Language* (UIL) in the X environment. Resources editing alter only interface presentation, there are no effects on the SDAI and callbacks do not need to be updated.

## 7.4 Data structure representation customization

Customization allows the designer to change the representation structure (it aims to alter the ordering and the assembling of objects) and to alter the navigation path through the application.

Again, these functions are standard in a *UIMS*. As an example of structure alteration, the designer can change the attributes representation order in an entity presentation, suppress an attribute presentation or choose an indirect representation in place of a direct presentation.

One cannot alter just anything. To sum up, only restrictive changes are allowed. It means that, once a component is initially generated, the elaborated component resulting from the user's modifications is always a subset of the original one. This restriction is due to the necessary coherence with the *SDAI*. The *SDAI* contains the data access functions which are coupled with the interface. Inserting, updating or deleting data through the *SDAI* forces the interface to respect the definition of the functions of the *SDAI*. This idea is similar to the notion of updatable view in a relational database system. Updates through a view are allowed providing the view definition takes into account some restrictive conditions.

Cross-reference of entities in the *EXPRESS application data schema* generates a navigation graph in the interface. It means that if an A entity is related to an B

entity, there is a way to go from the presentation of A to the presentation of B. The designer is able to modify this navigation graph by adding or removing access paths between interface components. He can also modify a *menu* item or a callback name.

## 7.5 Data structure representation migration

The data structure representation migration operates changes in the data structure itself. The third category groups together structural changes on the interface, such as adding a computed attribute or merging two components into one. This category is distinctive of both others because it is intended to solve two problems: the regeneration of the interface which preserves the designer's modifications, discussed in section 7.1 and the coherence with the *SDAI* discussed here.

Presentation components are strongly related to their data definition because the underlying behaviour of a data management application is meant to be the behaviour of the *SDAI*. So, structural changes made in a presentation component should affect the underlying *SDAI* functions. Moreover, we think that the need for a structural change in the user interface is in fact a need for an evolution of the data conceptual schema. For example, we consider that adding a computed attribute is not an interface modification but an application modification and that this modification should appear in the application data schema.

Considering all this, providing this kind of user interactions is very similar to providing schema evolution features in section 7.1.

## 7.6 Migration

### 7.6.1 Migration of the generated application

As discussed above, the building process should offer evolution operations for EXPRESS schemata of the application.

While the application is being designed, the application schemata are not instanciated (there are no application data). Therefore, we agree with D.Sanderson in [22] who uses the term of schema migration to distinguish it from schema evolution:

> *Schema evolution systems are designed to alter a schema as it changes over time while retaining compatibility to earlier definitions. A schema migration system allows radical changes to be made as a schema migrates from one format to another. Hence, a migration system will need operations that would be prohibited in an evolution system.*

Sanderson's works are implemented in a prototype system based on a version of a commercial migration system, ST-Developer [26]. Migration operations, called mi-

grators, should be accompanied by user interface changes. Using an open commercial migration system makes it possible to add functions to the migration operations.

So, a schema migrator operates (1) on the *EXPRESS application data schema* and (2) on the *user interface specification*. For example, the designer can decide to add the definition and the representation of an attribute. In such a case, he applies the operator *addAttribute* to the entity target. The *EXPRESS application data schema* is then updated, and an attribute representation is added in the *user interface specification*.

Some operators may imply some loss in the *user interface specification* because a default representation may be produced, erasing a customized one.

### 7.6.2 Migration of the building process

As discussed in 6.2, the building process is reflective.

Bringing alterations to the building process means that the designer changes the way that the *user interface specification* is generated. This is called meta-generation. The self-representation of the building process is made up of EXPRESS schemata. Changing the self-representation consists in evolving EXPRESS schemata.
The interface designer should be familiar with schema evolution, which is intensively used through the interface design process.
Hence, he should be able to use the same features at the meta-generation level, no longer acting on the application schemata, but on the *building rules schemata*.

**Schema migration** Evolving the *building rules schemata* at the meta-generation level can be dealt with as evolving the application schemata at the generation level: there are no instances to preserve. This means in fact that the designer must meta-generate before generating. The building process has to be tuned first so that the *building rules* are suitable to the designer's choices. Then the designer can use this same building process to design the desired interface.

**Instances migration** Another point of view is to allow the designer to meta-generate while generating. This means that while elaborating his interface, the designer may want to change the *building rules*. Building rules schemata are part of the *user interface specification schemata*. These schemata store instances which represent the interface under construction. This construction should be preserved. Keeping this interface (instances) while altering the rules (schemata) is no longer a schema migration problem but a database evolution problem: we need to migrate schemata and instances.

D. Sanderson in [23] describes the semantic effects of migration operations on EXPRESS schemata. Loss of semantics in the elaborated interface specification can

occur while using migrators. But the feature is interesting to maintain existing applications.

# 8 Related work

*Delphia Object Modeler (DOM)* [6] of *SLIGOS Inc* is closest to our work. *DOM* is a commercial tool. The conceptual method is *OMT* like [21], hence behaviour is specified by the designer. The main difference is that the behaviour must be specified by the designer. There is no way to elaborate the user interface, changes are made on the conceptual model and new interfaces are obtained by transformation.

# 9 Perspectives and further work

Some opportunities for future work are to:

- extend the default behaviour (not only the *SDAI*), to add general facilities such as cut/paste between objects, intelligent help or nested undo/redo.
- develop reflective aspects of the generated interface,
- expand meta-generation concepts and related migration.

# 10 Conclusion

This article has presented a building process of a data management application. The process reads an application data schema written in *EXPRESS* and produces a complete and working *user interface program* linked with a standard behaviour. The standard behaviour is made of the functions of the *STEP SDAI*.

The *building rules* are themselves made of *EXPRESS* schemata. The *interface specification* definition consists in the *building rules*, the *EXPRESS* dictionary schema and a core widget schema describing the user interface toolkit objects. So, the *user interface specification* is composed of all these schemata instances.

The designer can operate changes on the produced *interface specification*. Those alterations are carried over on the original schema through a reverse design process.

At a meta-generation level, the designer can also operate changes on the building process itself, using the operators of the generation level.

# References


[1] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Forth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, 1987.

[2] Grady Booch. *Conception orienté objet et applications.* Addison Wesley, 1992.

[3] J. Coutaz. *Interface homme-ordinateur, Conception et réalisation.* Dunod-Bordas, Paris, 1990.

[4] C. J. Date. *An Introduction to Database Systems, vol. 1.* Addison Wesley, Reading MA, 1990.

[5] H.R. Hartson and D. Hix. *Human-computer interface development : Concepts and systems for its management*, volume 5-93. ACM computing Surveys 21,1, March 1989.

[6] Eric Hubert. DOM : un atelier de génie logiciel objets pour la génération automatique d'applications interactives. *Génie Logiciel*, Juin 1995.

[7] ILOG. *ILOG VIEWS Reference Manual*, 1995.

[8] ISO TC184/SC4 CD 10303-22. *Part 22: Standard Data Access Interface*, 1994.

[9] ISO TC184/SC4 CD 10303-23. *Part 23: C++ Programming Language Binding to the Standard Data Access Interface Specification*, 1995.

[10] ISO TC184/SC4 IS 10303-1. *STEP Part 1: Overview and fundamental principles*, 1994.

[11] ISO TC184/SC4 IS 10303-11. *Part 11: EXPRESS Language Reference Manual*, 1994.

[12] ISO TC184/SC4, IS 10303-21. *Part 21: Clear Text Encoding of the Exchange Structure*, 1994.

[13] T. R. Kramer, K. C. Morris, and D. A. Sauder. *A Structural EXPRESS Editor.* NISTIR 4903, 1992.

[14] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *ECOOP/OOPSLA'90 Proceedings.* ACM Press, 1990.

[15] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA'87 Proceedings*. ACM Press, 1987.

[16] Masini and al. *Les langages à objets*. InterEditions, 1989.

[17] K. C. Morris. Architecture for the Validation Testing System Software. Technical Report NISTIR 4742, National Institute of Standards and Technology, Gaithersburg, Maryland, 1991.

[18] D.A. Norman and S.W. Draper. *User Centered System Design*. Lawrence Erlbaum Associates, 1990.

[19] John K. Ousterhout. *Tcl and Tk Toolkit*. Computer Science Division, University of California, Berkeley, CA 94720, 1993.

[20] D. Jason Penney and Jacob Stein. Class modification in the GEMSTONE Object-Oriented DBMS. In *OOPSLA'87 Proceedings*. ACM Press, 1987.

[21] James Rumbaugh and al. *Object-oriented Modeling and Design*. Prentice Hall, 1991.

[22] Donald B. Sanderson. A schema migrator for EXPRESS: Theory and practice. In *Fourth Annual EXPRESS User's Group, EUG'94, Conference Notes*, 1994.

[23] Donald B. Sanderson. Semantics effects of migration operations on EXPRESS schemas. In *EXPRESS User Group Conference Proceeding, EUG'95*, 1995.

[24] S. Schlaer and al. A deeper look at the transition from analysis to design. *Journal of Object-Oriented Programming*, Feb. 1993.

[25] B. Schneiderman. *Designing the User Interface, Reading MA*. Addison Wesley, 1990.

[26] STEP TOOLS INC., New York. *ST-Developer - ROSE Library Reference Manual*, 1992.