Generating Applications from Object Role Models

Betsy Pepels and Rinus Plasmeijer

Software Technology Department,
Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
{betsy, rinus}@cs.ru.nl

Abstract. We propose a generic strategy for generating Information Systems (IS) applications on the basis of an Object Role Model (ORM). This strategy regards an ORM as specifying both static and dynamic aspects of the IS application.

We implemented the strategy in a prototype tool, thereby using state of the art software technology. The tool generates IS applications with a basic functionality.

We regard our strategy as a first investigation of a new way to generate IS applications. Many open and sometimes far reaching research questions arise from this first exploration.

1 Introduction and Motivation

Data models like Object Role Models (ORM's, [1]) are used for a structured development of Information Systems (IS's). After a data model has been set up, it is translated to an implementation scheme, most often a relational schema. This scheme is subsequently the basis of the actual IS implementation.

Many tools exist supporting the development of data models. A substantial part of them automate the translation of the model to the implementation scheme. *Visio for Enterprise Architects* [2] is an example based on Object Role Modeling. To realize subsequently the implementation of the IS application, still a lot of development activities (coding) have to be carried out. This is costly and error prone.

We propose a generic strategy for *generating* IS applications on the basis of an Object Role Model. Automated generation of IS applications is valuable because it reduces development activities substantially, and hence errors and costs. Even more important, when IS development consists of development of models, the IS development process reduces to managing models, offering more control over it.

In our strategy we shift the view on the notion of an ORM. In the classical view, Object Role Modeling is a well-defined *method* for creating an ORM. The resulting ORM defines the data in the UoD and their constraints in a very formal way, thereby specifying a *static* description of its possible populations [3].

We consider an ORM also to specify *dynamic* aspects of its population. Using both the static and dynamic view on ORM populations, we develop a generic strategy to generate IS applications.

Generating complete IS applications is a pretty over ambitious plan that we certainly can't achieve. To start with, we aim only at giving a *Proof of Concept*, thereby using a very limited class of ORM's. The functionality of the corresponding generated IS applications is accordingly limited.

The core ideas of our strategy are presented in section 2.

As part of the *Proof of Concept*, we constructed a prototype tool. For the actual implementation of our tool, we use the lazy functional programming language Clean [4], developed and implemented by our group. We describe the tool shortly in section 3.

In section 4, we conclude on this first exploration of our new way of IS application generation. Furthermore, we elaborate on the many open and sometimes far reaching research questions that arise from it.

2 Obtaining a Running Application from an ORM

In this section we present the core of our strategy: how to obtain a running IS application from an ORM.

Limitations. In this first approach, we only aim at demonstrating that applications can be generated with our strategy. We use a limited class of ORM's: we do not yet take into account nominalization, subtypes and derived fact types. Furthermore, we limit the possible constraints to only uniqueness constraints (UC's) and mandatory role constraints (MRC's).

We do not cover retrieval functionality yet. In first instance we focus on generating the parts of an IS application that *change* data, in contrast to *retrieving* them. Earlier research has already pointed out that it is very well possible to define query languages directly based on ORM's [5].

Basic Plan. A running application is some definition (a "program") being executed. Hence, to transform some ORM into a running IS application, we have to derive a program definition from that ORM and subsequently execute it.

We consider a running IS application to have basically three ingredients: (1) a fact store (2) functionality (3) a (graphical) user interface (a GUI). We elucidate in subsections 2.1 through 2.3 how we obtain a definition of each of these three ingredients from an ORM. In subsection 2.4, we show how these three definitions are put together and transformed into a running application.

This approach is similar to that of the *Conceptual Information Processor* (CIP) [1]. Our strategy results in a concrete CIP that is inferred from the ORM.

2.1 Deriving the Structure of the Fact Store

In our strategy, we use an IS store that closely matches the structure of an ORM. This differs with the standard approach, in which an ORM is translated to a relational schema.

We associate with the objects and fact types of an ORM data structures capable of holding their population. Because labels do not have a population, we do not associate data structures with them.

With an *object*, we associate a data structure containing the population of the object. With an *n-ary fact type*, we associate a data structure containing a population of n-tuples. The elements of such an n-tuple depend on the roles being part of the fact type. If the role is played by an object, the corresponding tuple element is a *reference* (or *pointer*) to a member of the population of that object. By using references, members of populations can be shared. If the role is played by a label, the tuple element has just the same type as the label. In this way, the fact store resembles some kind of directed graph.

The aim of our different approach is twofold.

First, when generating IS applications from ORM's directly, we dot not want to bother about additional transformations to some operational mechanism, that only serves implementation purposes. So we prefer to have, at least conceptually, a structure of the store not all too different from the ORM itself. If really needed, necessary transformations can be added in a later stage.

The second reason is much more important. By translating to a relational schema, we would limit ourselves unnecessary. Relational schemas allow only some simple types like `Int` and `String`, and sometimes types like `Date`. By using our kind of store, we open the way for richer data types being stored, for instance collection types like `Set` and `Bag`, or special purpose types like `Message`, or recursive types.

2.2 Inferring Functionality

Normally, the functionality of an application is defined separately from the data model. It can for instance be defined by Use Cases. In a running IS application, functionality manifests itself in the user interface.

Conceptually, implementing functionality involves coding all kinds of transformations from the data entered through the user interface to the the actual relational schema, the latter being determined by the data model. These transformations should guarantee the integrity of the data stored.

In our strategy, functionality is not defined separately, but it is inferred (at least partially) from the ORM. This does not only save time and costs, but is also ensures guarantees automatically the integrity of the data stored.

Method. We take a bottom-up approach: we start with deriving properties of populations of ORM's (both static and dynamic) and arrive step-by-step at groups of data that are logically manipulated together by the end user.

Changing the Population of an ORM. The population of an ORM as a whole consists of the single populations of its objects and fact types. At all times, a population must obey the constraints of the ORM. A population may *change*: one or more changes may (simultaneously) be applied to one or more populations of object/fact types. But a change only may be applied if, after the change, the new population doesn't violate the constraints. So constraints determine the changes allowed to be made to a population. In other words, the constraints of an ORM determine the *operational* or *dynamic* behavior of its population.

Constraints. Normally, in an ORM constraints are used to express properties of the UoD. For this, a collection of standard constraints is available. Also general constraints may be used, often denoted as text. When implementing the corresponding IS application, a few of the standard constraints can be translated to constraints the RDBMS supports, for instance MRC's. The rest somehow has to be implemented by coding. General constraints have to be first interpreted by the developer and then this interpretation has to coded too.

We consider constraints to be *predicates*. Predicates are (*mathematical*) *functions* taking arguments and having as result a `Boolean` value. Here, the arguments are taken from the population of the ORM, and the result of the function must be `True`, otherwise the constraint is violated.

We regard constraints to be *orthogonal* to the objects and fact types of the ORM. Where objects and fact types define the structure of the population of the ORM, constraints limit the actual members of the population.

In our strategy, every constraint in an ORM is to be expressed as a predicate. The developer defines them using the language Clean. This code is used as part of the generated IS application (see below).

Standard constraints are defined once and can be reused subsequently. Every general constraint has to be expressed as a predicate by the developer. This might seem tedious, but there are advantages compared to the traditional way of working. Clean is a language with a very high expressive power, so a constraint can be expressed easier than for instance by coding it in SQL. Furthermore, the developer is forced to state very clearly the exact meaning of the constraint, leaving nothing to the interpretation.

Business Rules. In our vision, business rules are conceptually the same as constraints: they can also be regarded as predicates limiting the actual population of the ORM. Hence they could be added as general constraints to the ORM. There is however a big difference here. General constraints can with some effort be expressed as predicates. Business rules, and certainly the more complex ones, aren't expressed as predicates that easy. In fact, there are two problems here. The first one is how to formalize business rules at all. The second one is to express them as predicates. How to do this in general is subject of future research.

Logical Units of Work. Next, we concretize what a change to an ORM population comprises. A change (or *operation*) to the population of a *single* object/fact type is *implicit*. We recognize three basic operations: element(s) can be added (`add`), or element(s) can be updated (`upd`), or element(s) can be deleted from a population (`del`). Our plan is to infer from an ORM *groups of basic operations associated with objects/fact types* that, when applied simultaneously to populations of those objects/fact types, keep the population of the ORM as a whole valid. Such a group we define to be a *logical unit of work* (an *luw*).

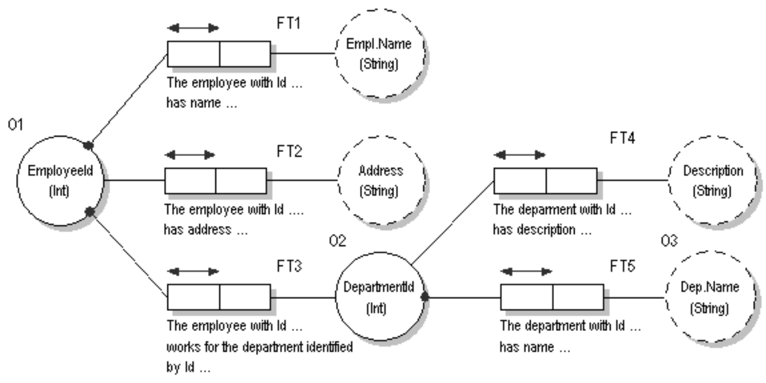

FT1
Empl.Name (String)
The employee with Id ... has name ...
O1
EmployeeId (Int)
FT2
Address (String)
The employee with Id .... has address ...
FT3
The employee with Id ... works for the department identified by Id ...
O2
DepartmentId (Int)
FT4
Description (String)
The deparment with Id ... has description ...
O3
FT5
Dep.Name (String)
The department with Id ... has name ...

Fig. 1. Example Object Role Model

Examples of Logical Units of Work. In the ORM in figure 1, {`add` on FT1, `add` on FT3, `add` on O1, `add` on O3} but also {`add` on FT2} are *luw*'s.

The term *luw* is chosen because from the ORM point of view, the group of operations is logical in the sense that, when applied simultaneously, it doesn't violate constraints. We will see below that also from the end user point of view an *luw* is logical too. Note that an *luw* applied as a whole keeps the population of the ORM valid; applying only a part of it might turn it invalid. Note further that *luw*'s are defined as operations on the ORM, but have an equivalent for the fact store.

Inferring Logical Units of Work. We use a generic algorithm to obtain *luw*'s from *arbitrary* ORM's. This algorithm is based on *abstract interpretation* [6]. When using abstract interpretation, in stead of reasoning in the concrete domain, it is reasoned in an abstract domain. The aim of abstract interpretation is to derive properties of the concrete domain, by simulating the behavior of the concrete domain in the simpler abstract domain. A detailed description of the abstract interpretation to obtain the *luw*'s is far beyond the scope of this paper and is subject of a separate paper.

The abstract interpretation algorithm takes as input the definition of the ORM and gives as result all possible *luw*'s of the ORM. The algorithm uses rules about constraints and populations, like "*if* a member is `add`'ed to a population of an object, *and* there is a MRC on a role played by that object *and* having a simple UC, *then* also a member *must be* `add`'ed to the fact type containing that role". The abstract interpretation algorithm starts with one object/fact type. It successively gathers all objects/fact types that are involved when changing the population, on basis of the rules. Together they form an *luw*.

Example. Starting with {`add` on FT5}, we find two *luw*'s: {`add` on O2, `add` on FT4, `add` on FT5}, and {`add` on FT5} solely.

Concurrently, the algorithm gathers every constraint that limits the populations of the objects and fact types of this *luw* into a *condition* that must hold if this *luw* is to be applied. This condition is expressed in the form of a function in Clean, taking as arguments the *actual population* of the ORM and the *actual data to be entered* in the ORM. This function is a *composite* of predicates that were earlier expressed by the developer as a Clean function.

From the abstract interpretation many *luw*'s result, some overlapping. It is up to the developer to point out which *luw*'s will be used for the IS application. *luw*'s can also be joined. Every *luw* used will appear in the running application as a means of manipulating data. Of course, the data the developer wants to be accessible, must be covered somehow in an *luw*.

Real life ORM's with a substantial number of fact types result in an enormous amount of *luw*'s. In this case it is not feasible to let the developer define by hand the set of *luw*'s to be used in the application. Some kind of automated support is needed. This is however unexplored terrain and subject of future research.

Access Model and Functionality. An *luw* involves a group of objects/fact types that is safely manipulated together. An end user has a different view on an *luw*. (S)he is interested in manipulating *data*, not in manipulating *fact types*. This means that the end user only manipulates the "leafs" of the *luw*: only the objects and labels of its constituting fact types.

This group of objects/labels associated with an *luw* we define to be the *access model* of that *luw*. Seen from the end user, an access model is a group of data that are manipulated together and are correlated in some logical way. They may for instance be presented together in a data entry window. This is the way the notion of functionality is correlated with an *luw*.

Example. For the *luw* {`add` on FT1, `add` on FT2, `add` on FT3, `add` on O1, `add` on O2} the access model is formed by O1, O2 and the labels identified by `Address` and `DepartmentId`.

The end user changes data in the form of the access model. These changes cannot be applied directly to the fact store. Therefore we additionally derive transformations (back and forth) between the definitions of the access model and the *luw*'s. These transformations again call the basic operations on the objects/fact types of the *luw*'s.

2.3 Generating the User Interface

To generate the the graphical user interface (GUI), we use the GUI Toolkit [7] that comes as a library with our development environment. The Toolkit works with models. To create an interactive application, it only needs a data model describing *what* what data are to be displayed and a model of *how* it should be displayed, also known as the *view*. The Toolkit is able to translate any change in the view into a change in the data model.

What should be displayed, we derived from an ORM in section 2.2: the access model. This can directly be used. *How* it should be displayed, is up to the developer. (S)he may define the appearance of the access model, for instance: the style, how data is presented (an edit box, drop down boxes) and the layout of the various elements.

2.4 Putting Together and Executing the Definitions

The three definitions we have obtained from an ORM (the structure of the fact store, functionality, the GUI) have to be put together and executed to get a running application. Each of the three definitions is brought to life in its own specific way.

Fact Store. The definition of the structure of the fact store is brought to life in the form of a *dynamic Object Space*. Initially, the Object Space is empty. Subsequently, for each object/fact type in the ORM, a corresponding data structure is created in the Object Space. The populations of these data structures can dynamically be manipulated by primitive access functions (`add`, `del`, `upd`).

We have chosen this approach because of the flexibility it offers: data structures can be added without having to stop and restart the application. In this way, we need only one fact store for storing both the facts defining the application and the facts of the application itself. Furthermore, this makes it possible to change (upgrade) the application while it is running.

Functionality. The definition of functionality is generated in in three parts: the access model, the conditions, and the transformations between the access model and primitive access functions. These parts are generated as functions and stored that way in the fact store.

To properly use these, we built a generic editor that works based on an access model. We create one window for each access model. To get an impression of this editor, see the screen shot of the running application in figure 2. The editor allows entering data in the form of the access model. The definition of the view is used for the actual displaying.

The editor contributes in maintaining the integrity of the data stored. Classically, when changing data in a system, the transaction is rolled back if it turns out that the integrity rules are violated. In our approach, the editor allows data changed in the window *only to be actually stored* if the conditions are satisfied. To check this, the editor calls the function defining the condition.

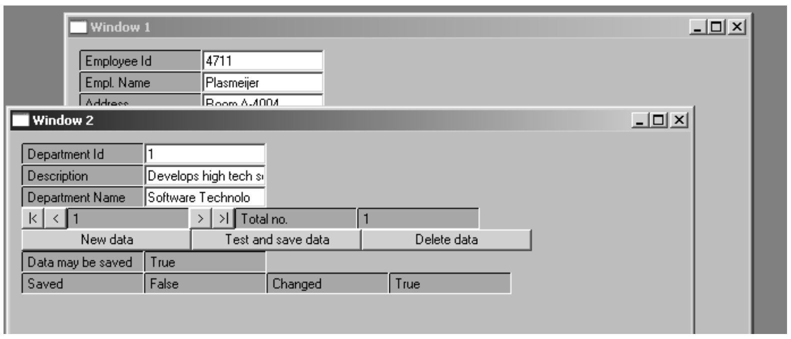
To put subsequently the data actually in the store, the editor calls the transformations between the access model and primitive access functions, which then are applied to the Object Space.

Graphical User Interface. Our GUI Toolkit creates the interactive part of the IS application, on basis of the access model and the view the developer defined.

3 Prototype Implementation

To test and demonstrate our ideas, we implemented a prototype tool. This prototype is in the first place meant as a research vehicle and not as a real IS application development environment.

For the actual implementation of our tool, we use the functional programming language Clean. The high abstraction level of functional programming languages enables developers to focus on architectural and algorithmical problems in stead



Employee Id	4711
Empl. Name	Plasmeijer
Address	Room A-4004

Department Id	1		
Description	Develops high tech s		
Department Name	Software Technolo		
1		Total no.	1
Data may be saved	True		
Saved	False	Changed	True

Fig. 2. Screen shot of the running application

of coding. We use its facilities for orthogonal persistence to write the fact store to disk [8].

Architectural Aspects. Every aspect of the IS application generation process is integrated in the tool. In the same environment, both the ORM is defined and the application is executed. The ORM is defined as facts and stored that way in the fact store. Both the intermediate and end results of the translation process and the data of the running application itself are facts as well and stored that way.

Working with the Tool. Using the tool, generating an IS application takes a number of steps. In the first step the developer records the ORM and constraints. In the second step the tool generates from the ORM a *standard* access model and view. In this preliminary version, the developer cannot yet determine for him/herself the composition of *luw*'s and not yet choose a view for them. In the last step, the whole is transformed into a running IS application.

To give an impression, in figure 2 a screen shot of the running application is given. There are two windows each containing the generic editor, one for entering data about employees, and one for data about departments.

4 Conclusion and Future Research

We outlined a generic strategy for IS application generation on the basis of a limited the class of ORM's. This strategy regards an ORM as specifying both static and dynamic aspects of the IS application. We use a fact store for the IS application that is directly correlated to the structure of the ORM. We infer the functionality of the application by generalizing the operational behavior of ORM's using abstract interpretation. This operational behavior is determined by the constraints and business rules of the ORM. The graphical user interface is generated by the GUI Toolkit of our development environment.

In our approach, the developer specifies constraints and business rules statically in the form of predicates. By our way of transforming the ORM into a running application, the dynamic behavior they imply is automatically accomplished.

We built a tool to test and demonstrate our ideas. For the implementation, we used the functional programming language Clean. Even starting from a very limited class of ORM's we are able to generate IS applications with a basic functionality.

Current Limitations and Future Research. The strategy presented here is a very first investigation of this way of IS application generation.

We started from a limited class of ORM's. First of all, we have to work out the strategy for complete ORM's. The abstract interpretation, which we couldn't present here, has to be worked out and described in detail, first for limited ORM's and successively for complete ORM's.

Our presentation of the strategy is an informal one. A more formal and generalized approach is needed, which should be based on, amongst others, operational semantics of ORM's.

The tool should be extended with the possibility of the developer defining the view on the access model.

Many open and sometimes far reaching research questions arise from this first exploration. A first and certainly not exhaustive list includes:

- Constraints and business rules play a central role in our strategy. They are to be expressed as predicates. Research is needed how (complex) business rules can be formalized at all, and how they can be expressed as predicates.
- Our strategy opens the way to have ORM's with labels and objects having richer types than types currently allowed, like collection types and recursive types. It is very promising to research how this is to be defined in an ORM, what the consequences are for IS development using ORM's, how it is to be incorporated in our strategy and what the consequences are for generating IS applications.
- The results of the abstract interpretation involve for practical ORM's large amounts *luw*'s. Research is needed for automated support to handle these.
- Real IS applications have various kinds of dynamic behavior. Behavior in ORM's arises from several sources, like the constraints and business rules and derivable fact types. Most probably, these do not suffice to obtain every desired dynamic behavior of IS applications. This implies that the ORM formalism has to be extended, for instance with explicitly defined flow.
- An IS application evolves. Conceptually, this means that its model changes and that the population has to be adapted to the new model. When IS applications can be completely generated on the basis of a ORM, automated evolution of IS's might become feasible. This is a very promising research theme.

References

1. Halpin, T.: Information modeling and relational databases: from conceptual analysis to logical design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
2. T. Halpin, K. Evans, P. Hallock, B. Maclean: Database Modeling with Microsoft Visio for Enterprise Architects. Morgan Kaufmann (2003)

3. ter Hofstede, A.H.: Information Modelling in Data Intensive Domains. PhD thesis, University of Nijmegen, The Netherlands (1993)
4. Clean home page. (http://www.cs.ru.nl/~clean/)
5. Bloesch, A.C., Halpin, T.A.: Conquer: A conceptual query language. In Thalheim, B., ed.: Conceptual Modeling - ER'96, 15th International Conference on Conceptual Modeling, Cottbus, Germany, October 7-10, 1996, Proceedings. Volume 1157 of Lecture Notes in Computer Science., Springer (1996) 121–133
6. Cousot, P.: Abstract interpretation based formal methods and future challenges, invited paper. In Wilhelm, R., ed.: Informatics — 10 Years Back, 10 Years Ahead. Volume 2000 of Lecture Notes in Computer Science. Springer-Verlag (2001) 138–156
7. Achten, P., van Eekelen, M., Plasmeijer, R., van Weelden, A.: GEC: a toolkit for Generic Rapid Prototyping of Type Safe Interactive Applications. In: 5th International Summer School on Advanced Functional Programming (AFP 2004). To appear in LNCS, Springer (2004) 262–279
8. Vervoort, M., Plasmeijer, R.: Lazy dynamic input/output in the lazy functional language Clean. In Peña, R., Arts, T., eds.: The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers. Volume 2670 of LNCS., Springer (2003) 101–117