Rapid Prototyping of Extended Entity-Relationship Models

Ellen Van Paesschen*
evpaessc@vub.ac.be

Maja D'Hondt**
mjdhondt@vub.ac.be

Wolfgang De Meuter*
wdmeuter@vub.ac.be
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

Abstract: The entity-relationship model is considered to be the standard for conceptual design of data in information systems and relational databases. *Extended* entity-relationship models provide extra concepts such as generalization, dependency and classification. In the context of object-oriented software development, such models are able to represent part of the functionality whereas more operational functionality is implemented in the object-oriented language. In the resulting object-oriented program, however, the original data model is often lost and the relationship and dependency constraints are not enforced. We propose an approach which consists of an *active modeling* phase and an *interactive prototyping* phase. The overall result is a model in which the elements have active links to fully operational implementation objects, which in turn are actively constrained by the model. We have developed a system that supports the manual steps in these two phases and fully implements the (semi-)automatic steps.

Keywords: Rapid prototyping, Extended ER models, object-orientation

1 Introduction

Nowadays, the entity-relationship model [3] is considered to be the standard for the conceptual design of the data in information systems and relational databases. The reason for this is twofold. On the one hand, the graphical format of entity-relationship diagrams is an intuitively understandable platform for both database or system users and developers, thus facilitating communciation between them. On the other hand, entity-relationship modeling is the starting point of a systematic development process for relational databases: there exists a clear, nearly one-to-one mapping between an entity-relationship diagram and a relational database schema [2].

Many extensions have been proposed to the original entity-relationship model [25]. In the context of object-oriented conceptual modeling, adding the generalization concept [20] was the most significant extension. This model is often referred to as the extended entity-relationship model (EER) [7]. Other EER modeling concepts are dependency, specialization, classification, and aggregation.

* Programming Technology Laboratory
** System and Software Engineering Laboratory

During object-oriented software development, an EER model is only able to represent part of the functionality. In this case, the object-oriented program is the most complete model for the other, more operational, functionality, which can be implemented in method bodies for example. However, the original data model is often lost in the object-oriented program. Moreover, the relationship and dependency constraints modeled in the EER diagram are not enforced in the object-oriented program.

We propose to install and maintain an *active link* between EER models and object-oriented programs. More specifically, our approach consists of an *active modeling* phase and an *interactive prototyping* phase. The first phase ensures that drawing EER modeling elements results in implementation objects being created that are continuously synchronized with changes to the model. Due to the "live" character of the link between model and code, code added to the implementation objects is preserved when the corresponding modeling element is changed. The second phase interactively converts these implementation objects into ready-to-use objects on which the modeled constraints are automatically enforced. This is entirely in line with the future directions for conceptual and EER modeling formulated by the inventor of EER models, Chen [4]. These directions include *executable active EER models, interactive modeling and computing* and of course *visual modeling.*

We have developed a system that supports the manual steps in these two phases, such as EER modeling, and fully implements the (semi-)automatic steps. As such, our system supports rapid prototyping of models.

This paper is organized as follows: in section 2 we start with EER modeling and our notation for all the modeling elements it provides. Next, active modeling, the interactive prototyping phase and how the modeled constraints are automatically enforced in the resulting implementation, are described. Note that we introduce our approach in an implementation-independent way: we describe it in terms of implementation objects, without specifying the object-oriented programming language's object grouping mechanism, for example, which can be either class-based or prototype-based. For our choice of implementation environment we refer to section 3. Section 4 summarizes related work while section 5 includes a conclusion and the future research directions.

2 Prototyping EER Models

Currently there is no real standard notation for EER diagrams. Most of the differences between notations concern how relationships are specified and how attributes are shown. In almost all variations, entities are depicted as rectangles with either pointed or rounded corners. The EER notation we use combines existing approaches: Chen's boxes [3], the relations of the crow's feet notation and the cardinalities of [7]. Additionally, the order of cardinalities is reversed, as in the Object Modeling Technique (OMT). Different colours denote the differences between entities and weak entities, and between simple, primary and derived attributes. We want to stress the fact that our new combined notation is merely a consequence of our choice of development platform (cfr. section 3). In figure 1 a small banking information system is modeled. A `Customer` for example, has a primary attribute `customerID`, and simple attributes `customerName`, `customerStreet`, and `customerCity`. `Customer` is in a many-to-many relation with `Loan` (role `borrows`) and with `Account` (role `accounts`), and in a many-to-one relation with `Employee` (role `banker`). `Payment` is a weak entity that is dependent of `Loan`. An `Account` can be specialized into a `SavingsAccount` or a `CheckingsAccount`.

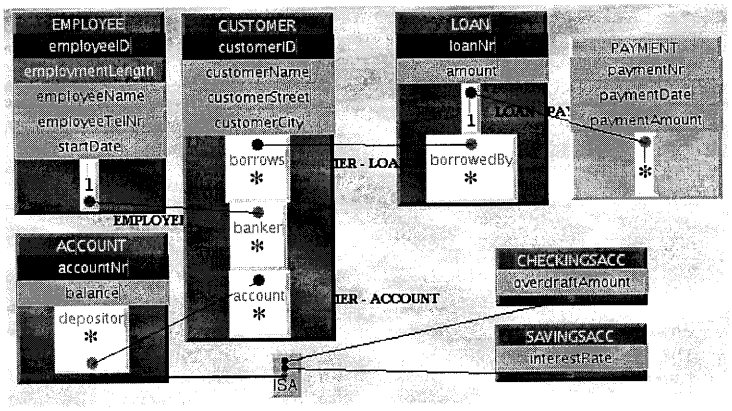

Fig. 1: EER diagram for a small banking information system

Other concepts that are not present in the example but provided by our approach include aggregation and categorization.

2.1 Active Modeling

In this first phase of our approach, the user develops an EER model in a graphical way. Since we fully support all the typical EER modeling elements, the user can simply employ a standard EER modeling methodology to construct the model. The contribution of our approach, however, is that in the background an object-oriented program is *automatically* created that reflects the EER model being constructed. More specifically, our EER modeling tool is implemented in the same object-oriented programming language, which means that EER modeling elements are represented as objects, which we refer to as *entity objects*. Based on these entity objects, corresponding objects are created that implement all the modeled attributes, roles and relations. We refer to such objects as *implementation objects*. While the user iteratively and incrementally adapts the EER model until it represents the intended system or domain at the conceptual level, the corresponding implementation objects are automatically updated. As such, the EER model becomes an *active model* with a live link to the corresponding object-oriented implementation. Afterwards, additional behaviour that cannot be expressed in EER models can be added to the ready-to-use implementation objects. For example, some behaviour is simply implemented as a method on an implementation object in the object-oriented programming language. More importantly, if the EER model is changed, this additional code is preserved in the implementation objects.

In order to achieve this, a set of mappings from entity objects to implementation objects has to be employed, as described in table 1. Note that implementation objects are often extended with extra information. This is necessary because the information is required in the interactive prototyping phase described in the next section.

2.2 Interactive Prototyping

The second phase of our approach is a *semi-automatic*, interactive prototyping process. When the EER model is complete our system has — based on the mapping in table 1 — created implementation objects containing extra information about the structure of

User action on EER model	Automatic action on implementation
Create new (weak) entity object	Create new corresponding implementation object
Add/remove primary/simple/derived attribute in entity object	Add/remove corresponding attribute in corresponding implementation object
Change name of attribute or entity object	Change name of corresponding attribute or implementation object
Add 1-to-1 relationship between two entity objects	Add attribute `1_to_1_relation_name1_name2` to each corresponding implementation object that contains the other object Add extra constraint information to the objects
Add 1-to-n relationship between two entity objects	Add attribute `1_to_n_relation_name1_name2` to each corresponding implementation object that contains the other object Add extra constraint information to relevant object
Add m-to-n relationship between two entity objects	Add attribute `m_to_n_relation_name1_name2` that contains the other implementation object
Add role to relation in entity object	Add attribute `role_in_m_to_n_relation_name1_name2` to corresponding implementation object that contains the name of the role
Add specialization from child entity object to parent entity object	Let child implementation object inherit from parent implementation object Add extra specialization information to child
Add categorization to entity object	Add extra categorization information to corresponding implementation object
Add aggregation to two entity objects	Create aggregation implementation object which refers to corresponding implementation objects
Make weak entity object dependent of to entity object	Add extra dependency information to corresponding weak implementation object

Tab. 1: Mapping from entity objects that represent EER modeling elements to implementation objects expressed in an object-oriented programming language

the corresponding entity objects. However, this information cannot be present in objects that are manually implemented. Therefore, we provide a strategy for creating ready-to-use implementation objects that do not contain this extra information but still adhere to the structure of the original entity objects. This strategy cannot be supported in a fully automatic way, because choices need to be made that depend on the preferences of the user of the program.

The strategy for creating ready-to-use implementation objects can be summarized by the following steps. Whenever we use the word *object*, we mean *implementation object*.

1. Create a new object based on an existing object.
2. For each categorization, ask the user into which parent object the new object needs to be categorized. Let the new object inherit the selected parent object. After implementing the categorization, it's no longer necessary to keep this information explicitly in the object. Therefore, we remove this categorization information from the new object.
3. For each specialization, ask the user into which child object the new object should be specialized. Add a reference in the new object to child object. Remove this specialization information from the new object.
4. For each relationship in which a new object has a *single* reference to another object, add an attribute to the former object representing the reference. Ask the user whether

to establish the actual reference. If so, initialize the attribute with a reference to the other object. If the new object is weak, its attribute has to be initialized with an actual reference. If a role name is present, rename the attribute to the role name. Remove the original attribute `1_to_..._relation_name1_name2` that was added during the active modeling phase in the new object.

5. For each relationship in which a new object has *multiple* references to other objects, add an attribute to the former object representing these references. Ask the user a default value for the number of actual references. Initialize the attribute with a reference to a container object of the required size. If the new object is weak, its attribute has to contain at least one actual reference. If a role name is present, rename the attribute to the role name. Remove the original attribute `..._to_n_relation_name1_name2` that was added during the active modeling phase in the new object.

2.3 Constraint Enforcement

After the interactive prototyping phase, ready-to-use implementation objects still contain some hidden information. This information is used to ensure that they satisfy two kinds of constraints *at all times*: (1) constraints related to dependencies between normal and weak implementation objects and (2) constraints about relationships between implementation objects. We explain below how both kinds of constraints are enforced.

- **Enforcing Dependents:** If an implemenation object is deleted, all the weak implementation objects that depend on it are deleted as well.
- **Enforcing Relationships:** When two implementation objects have a relationship in which the first one has a single reference (one or zero) to the second one, the uniqueness of this reference is enforced in two ways. First of all, our system ensures that no other attribute of the first object has a reference to another object of the second object's type. Secondly, we also ensure that only one object of the second object's type refers to the first object. If two implementation objects are in a 1-to-1 relationship, this is enforced in the two directions.

3 Implementation

As a development platform we selected the object-oriented prototype-based programming environment Self [28]. Prototype-based languages (PBLs) [16] can be considered object-oriented languages without classes. Self is closely related to the syntax and semantics of Smalltalk [9] but Self has no classes. The most interesting feautures of PBLs are *creation ex nihilo*, *cloning*, *dynamic inheritance modification*, *delegation with late binding of the* `self` *variable*, *dynamic parent modification*, and *traits objects*.

Self is a textbook example of a PBL and moreover, includes a mature programming environment. In the GUI, implementation objects are visually represented with pluggable outliners, constructed with morphs of the Morphic framework. This minimizes the distance between design and implementation: a developer can recognize the graphical design model in the visual representation of the implementation objects. Therefore, Self is considered to be an ideal candidate for extension with a conceptual analysis level: we made it possible to draw EER diagrams in the Self environment. Figure 1, contains in fact a screendump of the application. However, we consider the actual implementation details to be out of the scope of this paper. For more details on standard modeling and (multiple) inheritance in Self we refer to [26, 27, 28].

4 Related Work

Since the late eighties, it has been encouraged to combine (E)ER models and object-orientation (OO) [5, 21]. The (E)ER model is usually mapped onto objects to be used in object-oriented databases, and not onto "real" programming objects. However, there is no difference between these objects at the design level. Various approaches and techniques exist for translating EER into object-orientation. In [15] the gap between the ER and the OO model is bridged by introducing the category type and the possibility to define relations between types. Both [19] and [22] apply a set of mapping rules to transform EER diagrams into OO schemas. In [18] EER schemas are transformed into OO schemas by using a clustered form of the EER schema to improve understandability and to reduce complexity of conceptual schemas. In [8] a default mapping between EER diagrams and OMT diagrams is provided, i.e. from entity objects to design objects. These guidelines are then used to create or re-engineer object-oriented databases, but are also suitable to develop implementation objects. [17] introduces evolutionary ER schemas: the original ER schema is mapped to a version derivation graph which is in his turn mapped onto an object-oriented data model. In [11] clear steps are described to map an EER diagram into an object model, in order to re-engineer an existing database into an object-oriented database. In [10] a formal transformation scheme is provided to map ER schemas onto an object-oriented specification language, to be used for object-oriented databases. In [14] a formal calculus to tranform ER schemas into object schemas is defined.

Few of the approaches support automated mappings from (E)ER to objects. If they do, these applications generate a corresponding group of objects for an entire model or apply complex change management techniques for generating a selection of a model. This is opposed to our approach, which supports incremental generation *per-object*: one object is generated or synchronized for one EER entity.

Moreover, when mapping conceptual models in the (E)ER format to (design) objects, each object usually needs its own constructor and destructor method, with specific code for the dependency and relationships constraints [8]. Therefore, constraint enforcement is only supported at object creation time whereas in our approach it is supported at run time. Moreover, the methods responsible for creating (`copy`) and deleting (`delete`) implementation objects are generic, and shared by all implementation objects that were ever created for a corresponding entity object.

The SUPER system [6], based on the ERC+ model [24] (an object-based extension of the ER model), is claimed to supports definition, manipulation, and evolution of databases. Similar tools for visual data manipulation of object-oriented databases almost always involve a formal translation or compilation step to connect the conceptual view to the real objects. OdeView [1] for example, uses dynamic linking: every time an object needs to be displayed, dynamically loads the object file containing the appropriate display function.

To our knowledge, none of these tools implements an active link until the level of individual objects and attributes: often, when we change an attribute's name in the (E)ER diagram, the corresponding objects are not automatically updated but a new mapping will, in the best case scenario, provide the existing object with a new attribute. In our approach, there is no new attribute generated: the existing one is simply renamed. Moreover, to our knowledge, none of them allow the user to configure the objects interactively at runtime, solely based on the the static structure of the underlying EER model.

5 Conclusion and Future Work

In this paper we introduced a two-phase approach to actively link EER modeling elements to implementation objects. In a first phase a user draws an EER diagram while corresponding implementation objects are automatically generated. Modeled entities and corresponding implementation objects are continuously synchronized, but code added to the implementation objects is preserved during model changes. In the second phase, ready-to-use programming objects are created interactively: the user configures objects, at run time, based on the static structure of the underlying EER model. Moreover, dependency and relational constraints are enforced at runtime during the life time of the implementation objects. This approach has successfully been implemented in a prototype-based object-oriented environment, in which we integrated a grahical editor to draw EER diagrams.

For the moment, the EER entity object is bidirectionally linked to its corresponding implementation object. It is our intention to decrease this gap between EER and code by letting the EER objects really *become* the implementation object. This means that the EER entity is the visual representation of the implementation object in the background.

We will also extend the interactivity during the creation process. For mapping "is-a", 1-to-1, 1-to-n, and m-to-n relations at the conceptual level to real code, there exist a variety of implementation techniques. For example, there exist a series of patterns to implement different kinds of 1-to-1, 1-to-n, and m-to-n relations to code [23, 12]. We will offer the prototyper the choice of implementation technique while configuring the system.

Finally we might extend this forward engineering application with reverse engineering: changing the generated objects will influence the corresponding EER entities. In this way, we can implement a round-trip engineering [13] process between EER diagrams and prototypes systems.

Bibliography

1. Agrawal, R., Gehani, N.H., Srinivasan, J.: OdeView: The Graphical Interface to Ode, in Proc. of ACM SIGMOD '90, Int'l Conf. on Management of Data, pp. 34-43, Atlantic City, 1990
2. Batini, C., Ceri, S., Navathe, S.: Conceptual Database Design, an Entity-Relationship Approach. Benjamin and Cummings Publ. Co., Menlo Park, California, 1992
3. Chen, P.: The Entity Relationship Model - Toward a Unified View of Data. Massachusetts Institute of Technology, 1976
4. Chen, P., Thalheim, B., Wong, L.: Future Directions of Conceptual Modeling. Conceptual Modeling, 1997
5. Chen, P.: ER vs. OO. In Proceedings of the 11th Internaltional Conference on Entity-Relationship Approach, 1992
6. Dennebouy, Y., Andersson, M., Auddino, A., Dupont, Y., Fontana, E., Gentile, M., Spaccapietra, S.: SUPER: visual interfaces for object + relationship data models. Journal of visual languages and computing, 6(1), p. 27 - 52, 1995
7. Elmasri, R., Navathe, S.: Fundamentals of Database Systems. Addison-Wesley World Student Series, 3rd edition, 2000
8. Fong, J.: Mapping extended entity-relationship model to object modeling technique, in ACM SIGMOD RECORD, Vol. 24, No.3, pp. 18-22, 1995
9. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley, 1983

10. Gogolla, M., Herzig, R., Conrad, S., Denker, G., Vlachantonis, N.: Integrating the ER Approach in an OO Environment. In R. Elmasri, V. Kouramajian, and B. Thalheim, editors, Proc. 12th Int. Conf. on Entity-Relationship Approach (ER'93), pages 382–395
11. Gogolla, M., Huge, A.K., Randt, B.: Stepwise Re-Engineering and Development of Object-Oriented Database Schemata, in International Workshop on Database and Expert Systems Applications, Vienna, Austria (1998)
12. Génova, G., Ruiz del Castillo, C., Llorns, J.: Mapping UML Associations into Java Code, Journal of Object Technology, 2(5): 135-162, 2003.
13. Henriksson, A., Larsson, H.: A Definition of Round-trip Engineering, Technical Report, Department of Computer and Information Science, Linkopings Universitet, Sweden, 2003
14. Herzig, R., and Gogolla, M.:Transforming conceptual data models into an object model. In Proceedings of the 11th international conference on entity relationship approach held in Karlsruhe, Germany, edited by G. Pernul and A. Tjoa, 280–98, 1992
15. Kilian, M.: Bridging the Gap between O-O and E-R. In T.J. Teorey, editor, Proc. 10th Int. Conf. on ER-Approach, pages 445458, 1991
16. Lieberman, H.: Using prototypical objects to implement shared behavior in object oriented systems. In Meyrowitz, N., ed.: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Volume 22, p. 214 - 223, 1987
17. Liu, C.T., Chang, S.K., Chrysanthis, P.K.: Database Schema Evolution using EVER Diagrams. In Proc. of Intl. Workshop on Advanced Visual Interfaces, 1994
18. Missaoui, R., Gagnon, J-M., Godin, R.: Mapping an Extended Entity-Relationship Schema into a Schema of Complex Objects. In M.P. Papazoglou, editor, 14th Int. Conf. OOER, LNCS 1021, pages 204215, Berlin, 1995. Springer.
19. Nachouki, J., Chastang, M., Briand, H.: From Entity-Relationship Diagram to ObjectOriented Database. In Teorey, T., editor, Proc. 10th Int. Conf. ER-Approach, pages 459474, 1991
20. Navathe, S., Cheng, A.: A methodology for database schema mapping from extended entity relationship models into the hierarchical model. In The Entity-Relationship Approach to Software Engineering, G. C. Davis et al., Eds. Elsevier North-Holland, New York, 1983
21. Navathe, S., Pillalamarri, M.: OOER: Toward Making the E-R Approach Object-Oriented. In Proceedings of the 8th Internaltional Conference on EntityRelationship Approach, 1989
22. Narasimham, B., Navathe, S., Jayaramam, S.: On Mapping ER and Relational Models into OO Schemas. In R.A. Elmasri, V. Kouramajiam, and B. Thalheim, editors, Proc. 12th Int. Conf. on ER-Approach, pages 402413, Arlington, Texas, 1993
23. Noble, J.: Some Patterns for Relationships, in Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific 21), Melbourne, 1996. Prentice-Hall.
24. Parent, C., Spaccapietra, S.: About Complex Entities, Complex Objects and Object-Oriented Data Models, in Information Systems Concepts - An In-depth Analysis, E. D. Falkenberg, P. Lindgreen eds., pp. 347-360, North-Holland, 1989
25. Teorey,T.J., Yang, D., Fry, J.P.: A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship-Model. ACM Computing Surveys 18, 197-222, 1992
26. Ungar, D., Smith, R.: Self: The Power of Simplicity. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Volume 22., ACM Press, 1987
27. Van Paesschen E., De Meuter, W., D'Hondt, T.: Domain Modeling In Self Yields Warped Hierarchies, in proceedings of the ECOOP 2004 Workshop on Mechanisms for Specialization, Generalization and Inheritance, Oslo, Norway, June 14-18, 2004.
28. Self Home Page: http://research.sun.com/research/self/.