ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Rainer Manthey and Viacheslav Wolfengagen (Eds)

Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

Conceptual Data Modeling, Implementation Prototyping, Transformation, and Application Design -- An Animating Approach using RADD

Meike Albrecht, Margita Altus, and Martin Steeg

Published in collaboration with the
British Computer Society

Conceptual Data Modeling, Implementation Prototyping, Transformation, and Application Design - An Animating Approach using RADD

Meike Albrecht, Margita Altus, and Martin Steeg
Brandenburg Technical University of Cottbus
Computer Science Institute
Research Group for Databases and Information Systems

Abstract

This paper presents an approach to support the design of ‘dynamic’ databases where large amounts of data are updated frequently. The basis of our approach is formed by schemata of an extended entity-relationship model which are constructed in a modular way and can then be enriched by behavior specifications, functions, and view definitions. The goal is to derive the retrieval and update semantics from a given data schema as far as possible. The components of the database design toolbox RADD can be used to derive and analyze applications of the resulting information system, such that alternative EER representations with better performance properties can be automatically inferred, and visualized to the database designer. In this way, bottlenecks of the modeled system can be omitted – using the conceptual view to the database, and a logical formalism for requirements specification and database tuning.

1 Introduction

A well-known problem of information system design is that impacts of the earlier modeling phases (conceptual, logical, and physical design) for the subsequent phase and the modeling’s effects for database maintenance are almost not considered. Given a today’s graphical database design, such as an entity-relationship schema or an object model, operations have mostly been considered by either presupposing them (select, insert, delete, update in some ER design tools) or declaring their names and providing their signatures (object models). An advantage of object models - that must not be ignored - is that with the design, the structuring of classes and methods is already given for the later implementation. However, entity-relationship as well as object models do in general not analyze the data, module, and method structuring’s impacts according to *processing properties* of the resulting system. These properties describe the reaction of the database system to operations such as storing, updating, and retrieving the data or objects.

Most tools coerce the designer into a restriction of their modeling view to graphical presented inheritance keys, declaration of dummy methods, or generation of content-less templates for triggers and database procedures. These give the designer the impression that many possibilities are available, but make a *good* design of graphical data or object models quite complicated. More concrete: These schemata are – without adaptation and restructuring – not appropriate for database maintenance. Behavior, performance, and processing requirements often coerce the designer into a completely redesign of the schema¹.

There are several reasons why processing information should be included in the earlier phases of database design:

¹Notice that object models even make these design problems worse, since they consider neither efficiency nor performance.

1. *Low storage complexity* is an objective of conceptual and logical design. *Efficiency* is the main database processing requirement. The latter goal is often in contradiction to the former which is realized by normalization.
2. Normalization is the typical approach to minimize storage complexities and operation anomalies. But, normalization algorithms may derive several schemata from the same attribute set and functional dependency set, [27]. Thus, we need to know which one is optimal.
3. Structural well-designed schemata often imply operational bottlenecks. For different DBMSs different tuning principles are appropriate, [10, 17, 22]. Database structures for the most efficient implementation often cannot be derived directly from the given. Hence, *external views* may not be supported by the '*tuned*' *internal schema*.
4. In general, the database system has to verify all the static constraints after each update. In most cases however, not all the static constraints have to be verified, but still this verification is a very time consuming activity, which we want to reduce as much as possible. [15]

Therefore, modern database design approaches advice to include processing information as well. For instance, in [12] the authors argue why processing requirements should be already considered at early design steps: *conceptual and logical design often generates a database schema that cannot easily be modified once the database is implemented*.

RADD implements a strategy which incorporates requirements analysis and design impacts for the later database maintenance already into conceptual design. This makes database development more transparent, flexible, and consistent, but does not require the conceptual designer to refer to logical and physical design aspects, since these are inferred by the toolbox. Logical and physical design aspects are highlighted in the conceptual design to notify the designer of possible mistakes.

The paper is organized as follows. Section 2 shows the graphical design possibilities for a modular construction of EER schemata within RADD. Section 3 gives an overview on how the graphical design is enriched by behavior specifications, function and database application module definitions, and, how physical performance properties are derived from the schema. Section 4 shows how our approach can be used for database system realizations. A special application that is discussed here, is a database system for a *Companies* departments, projects, and staff. This is not a typical application for a 'dynamic' database since update operations for this kind of application may occur relatively rare. However, the approach can be easily extended and works fine for dynamic databases too, e.g. an information system concerning the *generation and distribution of notices for air traffic control*. The article concludes with prospects for the future.

2 Graphical Database Design

Analyzing known design strategies, primitive steps can be extracted. *These primitives are the basis for the strategy support in RADD*. The designer can choose his own strategy and compose the strategy from primitives. The system RADD supports this choice based on the user model, user preferences, and characteristics of the application. Thus, the designer can switch among bottom-up, top-down, modular, inside-out, and other strategies. The *controller* derives graph grammar rules for the maintenance of consistency and for guiding explanations. Strategy consultancy and error location are included in the user guidance tool. Apart from the primitives, each design strategy is related with a set of *checkpoints*.

The designer applies a heuristic search method, such as the A^* algorithm ([23], pp.160), using an estimation function. A normal user of a database design environment can be supported by implementing such search techniques and by providing *evaluation functions* and *design primitives* (abstract operators). The evaluation function estimates what the most *plausible* next design step according to the design goal is, and it is based on several pieces of evidence: populations of data items, cardinality constraints, inclusion and exclusion constraints, additional functional, afunctional and multivalued dependencies, i.e. semantics acquired and accumulated during earlier design steps and new discovered semantics including retrieval and update semantics. The plausibility

function of the evidence theory has been proposed to model the accumulation and evaluation of different pieces of evidence to support the *inductive design decision process*, [4]. The term design strategy is used as a synonym for search method.

The variety of different strategies is based on the dimensions of database design. Classical design strategies are well known such as top-down, bottom-up, mixed, inside-out, and modular design strategies, [24, 25, 26, 5]. The top-down and bottom-up design strategies are uncontrolled. The mixed and inside-out design strategies incorporate control functions: a skeleton schemata or semantic neighborhood, respectively. They are used as uniform-cost procedures in a kind of best-first ordered search. Each design strategy is characterized by the following different approaches (dimensions of database design), [24, 25, 26]:

- *The design direction:* The bottom-up design uses composition operations, the top-down design uses decomposition operations.
- *The control of the design:* Inside-out design strategies use a neighborhood function to select the next concept.
- *The modularity of the design:* In analogy with the modular design, the mixed design starts with the skeleton design and is then refined; the view-oriented design is a special case of modular design.

Therefore, a design strategy is composed of different approaches.

The database design process is more a *process of construction* than a process of mapping knowledge of the application. The design process is accomplished by design steps. Each design step is based on a *design primitive* and may require consistence checks. Design primitives are mainly characterized by the following operations: *composition*, *decomposition*, and *extension*, [24, 25, 26].

At present, graphical design systems hardly support the designer. Design systems of the future have to be flexible. Therefore, they have to be configurable and adaptable to the designer and to the application based on design rules. Designers prefer to reuse design modules and construct new design modules on the basis of other modules. If this is not possible they need rules to start design module construction from scratch. To support this style of working a design environment should include global application knowledge, knowledge about designers, design expertise such as knowledge about design strategies, and the design model. Thus, a complete, correct, and minimal set of atomic design primitives, a library of design modules which have been constructed based on the atomic design primitives, and design normalization have to be available in the design environment for the convenience of the conceptual designer.

An operational framework is included in our database design environment, and *guidelines* for the construction of design modules on the basis of atomic design primitives. Especially, the way to specify the content of database modules consistently and graphically. In the sequel, the term *unit* is used as synonym for database design module. This style-of-working composes the new design methodology *design-by-units* that incorporates information hiding. Design primitives are considered to be transformations on the design database.

The system is based on the RADD extended entity relationship model² which allows the user to specify graphically the structure of an application, the integrity constraints which are valid in the given application, operations, and transactions which are necessary for the given application. This extension requires an easy-to-handle and advanced support for graphics. Furthermore, the editor enables the designer to visualize the complete design information.

The user interface and the explanation component are adapted to skills, abilities, and intentions of the database designer. This tool allows customization of the user interface and the explanation component. [2, 3, 4, 27, 11, 12]

3 Database Design Optimization

Some authors share the opinion that the structure- and cost-based evaluation of the performance of database update operations during the conceptual design phase has mostly been neglected, e.g. [6, 15, 14, 10]. RADD

²The RADD extended entity relationship model is close to the HERM defined in [26].

Figure 1: Initially modeled *Company* schema.

allows one to observe the conceptual database schema for inconsistencies and probable performance bottlenecks. We use the conceptual schema to

- (a) derive internal database representations,
- (b) compute on the internal representation behavior and performance properties of transactions, and
- (c) discover aspects for more advantageous conceptual design,

in order to omit problems of database - implementation tuning – which often destroys or confuses semantic dependencies. The target of our approach is to develop a conceptual database schema that supports an efficient database application.

Most times insufficient processing requirement fulfillments lead to later database tuning, see for example [21, 17]. Processing requirements already influence the design and realization of commercially used information systems, but, the design of a new information system or database is rarely related to performance aspects of the system which is under construction. We now show how performance properties can be inferred once a schema (entity relationship schema or object model) is given, and how obvious bottlenecks of the designed system are used to restructure and improve the schema.

3.1 The Database Design Problem

Assume, the user has modeled a *Company* schema comprising entities *Employee*, *Department*, and *Project* and relationships *manages*, *works_on*, *works_for*, *supervises*, and *controls*, Figure 1. It is a not-so-small application. Also, it has been analyzed by other authors, e.g. [13, 27]. The initial schema has been designed with a top-down strategy, but without an explicit set of functional dependencies. Functional dependencies are conceptualized by *many-to-one relationships* between the object classes. To determine the functional dependencies that hold for the schema we have to consider carefully what the object classes (entity/ relationship types) mean. In this sense, dependencies are actually assertions about the real world. These assertions can be validated by examples using a component of RADD: semantic acquisition, Figure 2, [2].

There are inclusion constraints which are *not* represented implicitly in the *initial* schema graph:

Start)						
Quit)						
	Department.Name	Department.dnumber	Employee.Name	Employee.ssn	manages.StartDate	works_for.phone
1	Engineer.	1	Walker	135790	11-07-96	+49-355-13579
2	Laboratory	2	Mueller	975310	01-09-95	+49-355-97531
3	Computing1	3	Schulz	913570	01-01-96	+49-355-91357
4	Computing2	4	Newman	753190	01-01-96	+49-355-75319

=====						
Funkt. and afunkt. Abhaengigkeiten =====						
=====						

aus dem Dialog	Department.Name	---->	Department.dnumber
aus dem Dialog	Department.dnumber	---->	Department.Name
aus dem Dialog	Department.dnumber	---->	manages.StartDate
aus dem Dialog	Employee.Name	---->	manages.StartDate
aus Schluesse	Employee.ssn	---->	Employee.Name
aus Schluesse	Employee.ssn	---->	works_for.phone
aus dem Dialog	works_for.phone	---->	Department.dnumber
aus dem Dialog	works_for.phone	---->	Employee.Name
aus dem Dialog	Department.Name, Employee.Name	---->	works_for.phone
aus dem Dialog	Employee.Name, manages.StartDate	---->	Department.Name
aus dem Dialog	Employee.Name, manages.StartDate	---->	Department.dnumber
aus dem Dialog	Employee.Name, works_for.phone	---->	Employee.ssn

 Press any key !

Figure 2: Semantic acquisition of dependencies between the concepts *Department*, *Employee*, *manages*, and *works_for*.

- (1) If an employee *supervises* a project then the department *for* which he must *work* is *equal* to the department that *controls* this project.
- (2) If an employee *supervises* a project then the set of projects he is *working on* includes at least this project.
- (3) If an employee *manages* a department then at least this employee *works for* the same department, i.e. all managers are employees in the same department in which they work.

$$\forall e_1, p_1, d_1 : \text{supervises}(p_1, e_1) \Rightarrow (\text{works_for}(e_1, d_1) \Leftrightarrow \text{controls}(p_1, d_1)) \quad (1)$$

$$\forall p_1, e_1 : \text{supervises}(p_1, e_1) \Rightarrow \text{works_on}(e_1, p_1) \quad (2)$$

$$\forall d_1, e_1 : \text{manages}(d_1, e_1) \Rightarrow \text{works_for}(e_1, d_1) \quad (3)$$

3.2 Considering Information System Dynamics

Knowing the information system dynamics supports the inductive design decision process. That means, we like to determine which parts of the schema need to be strongly normalized, and, which parts or sets of structures can be collapsed according to improvement of retrieval performance. Information system dynamics can be derived in part from data item population aspects which are acquired for the structures of the designed schema in RADD.

```

Tuple number for Type 'Employee' >> 760
Tuple number for Type 'Department' >> 4
Tuple number for Type 'Project' >> 17
Tuple number for Type 'manages' [4] >> 4
Tuple number for Type 'works_on' [>=760] >> 875
Tuple number for Type 'works_for' [760] >> 760
Tuple number for Type 'supervises' [17] >> 17
Tuple number for Type 'controls' [17] >> 17
    
```

In contrast to traditional textbooks (e.g. [28, 27, 11, 16]), collapsing structures, e.g. an entity structure and a relationship structure, can improve retrieval performance and insert/ delete/ update performance as well. For the considered *Company* schema and the data item population aspects (numbers of tuples), the internal schema containing an *Employee* structure with a repeating group of *Project*-references has better retrieval performance and update behavior properties than one with a separate *Employee*, *works_on*, and *Project* structure, respectively since we have only 1.2 as much *works_on* instances as *Employees* and we can infer that most of the *Employees* work on exactly one *Project*. It is clear that whether operational behavior is really a bottleneck of the designed schema depends upon the frequency and priority of the mentioned operations:

- If insert is an often required operation for *Employee*, *Department*, *works_for*, and *works_on* then the insert operation creates a crucial bottleneck.
- If it is necessary to delete frequently an *Employee*, *Department*, or *manages* record then the delete operation has higher complexity.
- The update operation is inefficient for the entity structures *Employee* and *Department* and the relationship structure *works_on*.
- The retrieval operation for *works_on* is relatively complex.

Therefore, we could derive the following arguments.

1. If we need frequently the above mentioned operations then the integration of explicit constraints in the schema is necessary.
 - The relationship *manages* is a subtype of *works_for*, identifying the special role of department managers.
 - The relationships *supervises* and *works_for* are related, i.e. the *Employee* who supervises the *Project* must work for the *Department* that controls the *Project*.
2. The collapsed *Employee/works_on* structure on the implementation level (a repeating group in a hierarchical or relational database, a set-typed attribute in a network or object database) gives us the advantage that we can omit a join operation. If the entity occurrence sets *Employee* and *Project* are not changed frequently the aforementioned relationship structures should be grouped in the case in order that their update operations get a very high frequency and priority. The retrieval operation for type *works_on* should be considered as important since information according to projects and their staff is frequently required.

3.3 Adding Behavior Properties to the Graphical Design

In RADD, the designer has to specify error prevention options for each transaction *separately*. This improves the design and the maintenance of design specifications since normally, there is an arbitrary number of repair reactions, but the following four cases (behavior properties³) identify very common patterns coping with data inconsistencies in the running database.

1. *RESTRICT*. Cancellation of the transaction at appearance of data inconsistencies.
2. *CASCADE*. Invoking *repair actions* at appearance of data inconsistencies. (e.g., missing or deleted items of the parent structure.)
3. *SET NULL*. If the item of a parent structure for which still child items exist is deleted (or updated), then the corresponding child references are set to “null” (iff “null” is allowed).

³*Behavior properties* are additional programming options which form an important part of the entire behavior specification coping with possible errors caused by data inconsistencies in the running database

4. *SETDEFAULT*. On insert of some child item use a default value as reference to the occurrence set of the parent structure. If the item of a parent structure for which still child items exist is deleted (or updated), then the corresponding child references are set to the default value.

We have *default behavior rules* which are compiled into the conceptual database optimizer of RADD, and in the user's specification frame, *general rules* which are identified by a 'General' section, and *special rules* for special integrity constraints ('Special' section). Default behavior rules' specification frame for a part of the *Company* schema:

General:

```
On insert restrict;
On delete cascade;
On update parent
  IF nullable(child)
    THEN set null
    ELSE set default FI;
On update child cascade;
```

Special:

```
On insert("manages",m)
  IF Not (m.Employee in "Employee")
    THEN insert("Employee",s) FI;
For Reference "manages" TO "Employee":
  On delete restrict;
```

3.4 Adding Functions to the Graphical Design

RADD allows *behavior specifications* with the help of user-defined database functions and IF-THEN-ELSE statements, such that transactions can be specified programmatically. The conceptual database optimizer of RADD provides a functional language (*Csl*, *conceptual support language*⁴) specification interface with the following elements:

- *select*, *insert*, *delete*, and *update* denote database operations – Csl uses operation terms⁵; the conceptual support language also considers group and aggregation functions, such as *min*, *max*, *count*, *sum*, *avg*, etc.;
- *entity*, *relationship*, *cluster*, *component*, *tcomponent*, and *compatible* denote schema operations which are used to test the data context;
- *highcomplexity* denotes a parameterized operation which evaluates database operations;
- *attrsize* denotes the number of attributes that belong to one structure;
- *group*, *separate*, *nest*, *unnest*, and *clusterize* denote schema transformations;
- "+", "⇔", "*", "/", "~" (unary minus), and "^" (string concatenation) denote primitive operators, "=", "<", "<=", ">=", ">", and "<>" denote comparison operators, and ":=" denotes the assignment operator.

Values, i.e. constants, can be defined within Csl by:

$\langle valdecl \rangle ::= \text{"val"} \langle name \rangle = \langle val \rangle$

and functions are defined by:

$\langle fundecl \rangle ::= \text{"fun"} \langle name \rangle \langle par \rangle = \{ \mid \langle par \rangle = \langle val \rangle \}^*$

where $\langle val \rangle$ is a constant, a Csl-defined value or function, or a Csl expression:

⁴Essentially, Csl re-implements a subset of the Standard ML of New-Jersey functional language, and adds database operations and schema operations, where schema operations are schema transformations, schema tests, or evaluation operations, respectively.

⁵e.g. *insert(Employee)* or *insert(Employee, (Name="Victor H.", Bdate="11-01-59", Salary=54000,...))*


```

< val > ::= < const > | < name > |
    "fn" < par > ">=" < val > { " | " < par > ">=" < val > } * |
    "let" < decl > "in" < val > { " ; " < val > } * "end" |
    "(" < val > { " , " < val > } * ")" |
    "ref" < val >

```

In Csl, the symbol "fn" introduces the definition of a function, which is handled as a value only, i.e without name or identification, respectively. Based on the given ER schema, the type of all structures is assumed "struc", e.g.:

```

CSL> Employee;
it : struc = Employee

```

However, the concrete structure of *Employee* which is maintained by RADD, e.g.

```

{ Name: { Fnames: SET ( Fname: char(15) ),
          Lname: char(20),
          Title: char(15) null },
  Bdate: date null,
  Ssn: decimal(9) unique,
  Address: char(30) null,
  Sex: char(1) null,
  Salary: decimal(6,2) null }

```

is considered if Csl expressions (functions, values) that refer to *Employee* are evaluated.

Besides "struc", Csl types are "view", "module", "bool", "int", "real", "number", "date", "string", "data_schema", the generic types "tuple", "list", and "set", and "function". Csl maintains the RADD SQL-2 types {char, float, decimal} by {string, real, number}. However, the information of the database attribute types, e.g. char(15), is not lost but only used internally for export of the database schema.

Csl values can be used to specify "select" expressions, such that we can define a parameterized function identified by *avageolder* that evaluates the average age of all *Employees* whose age is older than *year* by:

```

CSL> fun avageolder year =
>     select avg(truncyear(today() - Bdate))
>     from Employee where truncyear(today() - Bdate) >= year;
avageolder : int -> real = <function>

```

Here, the Csl expression is compiled into an internal representation and the type of the internal representation is displayed. Csl implements type inference that raises exceptions on typing mistakes.

```

CSL> fun avageolder rel year =
>     select avg(truncyear(today() - Bdate))
>     from rel
>     where truncyear(today() - Bdate) >= year;
Warning: rebinding avageolder
avageolder : struc -> int -> real = <function>
CSL> avageolder Employee;
it : int -> real = <function>
CSL> avageolder Department;
TypingBug: cannot apply avageolder to Department

```

Unlike *Marchiavelli* [7], Csl's primary intend is not to maintain the database, that is to instantiate a database schema and then to evaluate select expressions. The primary intend is rather to have a mapping of which operations are required – and to provide an interface and a language to specify those requirements. For this reason, the data which have previously been defined by the RADD component *semantic acquisition* are used by the Csl interface as well to animate application module declarations

3.5 Providing Database Views and Application Module Declarations

Csl provides view definitions based on select expressions, e.g.:

```

CSL> view Department_Manager =
>     select Department.all, Employee.works_for Manager, Employee.mStartDate
>     from Department,Employee
>     where Department.dnumber = Employee.works_for
>           and Employee.mStartDate is not null;
Department_Manager :
( dnumber: integer, dname: string, location: string, Manager: integer, mStartDate: date )
= <view>

```

A Csl module is defined as follows:

```

< moddecl > ::= "class" < modname > [ ":" < modname > { "," < modname > }* ] "="
    [ "structures" { < struc > [ "=" < struc > ] }+ ]
    [ "knows" { < modname > "." { "all" | < name > } }+ ]
    { < viewdecl > }*
    { < iddecl > }*
    { < valdecl > | < fundecl > | < moddecl > }*
"end"

```

where a module is introduced by the keyword "class".

We do not distinguish between modules and classes, a module is considered as a class with exactly one instance. In the Csl interface to the user, we currently do only support the specification of modules without instances, that are proper *modules*. Also, in the Csl interface, the part for the inheritance specification '`[":" < module > { "," < module > }* ]`' is currently not enabled.

In analogy with Csl value and function definitions, Csl modules are compiled into an internal representation which makes the module code executable.

Nevertheless, the structures of the graphical design environment are maintained as modules with instances, that are *classes*. The arrows of the graphical design, such as "is-a" connections or, generally, the references of relationships to entities, relationships or clusters, are represented as Csl inheritance relations. We also plan to allow the user to specify (small) modules with instances, e.g. for database administration purposes, like menu button labels and so forth. This is the reason for that we use the keyword "class", to introduce the specification of classes and modules as well. An example of a Csl module is:

```

CSL> class Demo =
>     structures Employee Dept = Department works_for
>     view manages = select Employee.Name.Lname ename, Dept.Name dname, mgrStartDate
>     from Employee, Dept, works_for
>     where works_for.Employee = Employee
>           and works_for.Department = Dept
>           and not mgrStartDate is null
>           and mgrStartDate <= today()
>     val showDeptManages = fn dept => select * from manages where dname = dept
> end;
Demo : ( manages : ( ename: string, dname: string, mgrStartDate: date ) = <view>
      showDeptManages : string -> ( ename: string, dname: string, mgrStartDate: date ) set
      = <function> ) = <module>
CSL> open Demo;
manages : ( ename: string, dname: string, mgrStartDate: date ) = <view>
showDeptManages : string -> ( ename: string, dname: string, mgrStartDate: date ) set
      = <function>
CSL> showDeptManages "Engineering";
{ ( "Walker", "Engineering", "11-07-96" ) } :
      ( ename: string, dname: string, mgrStartDate: date) set

```

Csl modules are specified logically by use of conceptual elements, but they navigate the physical database which, in general, has a structure far different from the conceptual schema. In this way, Csl modules can be looked at as something like database views which provide navigation, inserts, deletes and updates to a physical

realized database. This user-interface approach is a combination of the *CoDO* approach of [19, 20] and some previous works of the authors. Here, the general idea is to provide the user with facilities which allow him to implement and maintain the database by a high-level conceptual and logical language, respectively, but to not (necessarily) confront him with low-level implementation details ⁶.

3.6 Automating the Design Transformation and Visualizing Database Transactions to the Conceptual Designer

As mentioned above, *group*, *separate*, *nest*, *unnest*, and *clusterize* are transformation primitives, which are used to automatically generate an internal database schema. The transformation – in order to provide a conceptual schema review – is driven automatically by *rules*. An example for such a conceptual schema to internal schema (in this case, relational schema) transformation rule is shown in Table 1. The symbol *MaxRepGrpSize* is a global constant, used as reference to decide whether internal structures are to be grouped (collapsed).

The generated internal schema is a different view to the database. Here, structures have been grouped, splitted, or restructured, in order to derive efficient database applications (refer to [18, 19, 20]). The Cottbus Conceptual Database Design Optimizer *CoDO*, which is part of the RADD toolbox, has a generic cost model and is using rules, to infer typical database transactions and derive performance properties for the conceptual schema. As we will see below, the database designer has the possibility to add new rules. It is also possible to change existing transformation and transaction rules or to choose a subset of all rules, to perform conceptual schema reviewing and optimization.

The applied transformation actions do not only refer to the transformation rules but also to the transaction and “behavior” properties (reaction patterns) which are specified for the schema. So, if there is a transformation rule which says to group two structures of the entity-relationship schema to one internal structure, but there is also an ON-INSERT-CASCADE rule between the two structures, then the structures are not grouped, since we assume that the user wants to see the repair reactions which are necessary for integrity maintenance between these structures.

Example. Assume, the database designer specifies an ON-INSERT-CASCADE rule for the reference from *works_for* to *Employee*:

```
CSL> For Reference works_for to Employee: On insert cascade;
Adding 1 new rules to schema Company.
```

Now, RADD assumes that the designer does not want the structures *Employee* and *works_for* to be grouped. According to this, Figure 3 shows how transactions and their costs which are evaluated on base of the internally generated schema, are presented to the database designer.

3.7 Conceptual Database Design Tuning

The performance properties of the internal schema are used to optimize the conceptual schema since performance bottlenecks which are recognized on the internal schema are often based on bad design decisions which are made conceptually. An example for an optimization rule is shown in Table 2. Using this rule-based

⁶In this context, “low-level implementation details” means to be familiar with options like *kind of storage structure*, *block sizes* or *rollback segment configurations*, that are provided by most DBMSs. It does *not mean* to be completely unexperienced with database maintenance, for example, to have never worked with a language like SQL.

Table 1: Design optimization rule - based on the structure.

$$\frac{\text{component}(R_1, R_2) \wedge ((1 = m = n) \vee ((1 < m \leq n) \wedge \text{attrsize}(R_1, (m, n)) < \text{MaxRepGrpSize}))}{CC(R_1, R_2, (m, n))} \rightarrow \text{group}(R_2, R_1, (m, n))$$

TheReVIEWer: Company				
COST-STATISTICS	INSERT	DELETE	Update of Employee ::=	
			UPDATE(Employee) => 27,59 +	
			0,5 * RETRIEVE(works_for)=4,37	
Department	122,71	87,24	0,25 * UPDATE(work	Update of manages ::=
Employee	173,68	131,68	0,5 * RETRIEVE(man	UPDATE(manages) => 22,24 +
Project	67,19	35,7	0,0 * UPDATE(manag	0,5 * RETRIEVE(Employee)=5,62
supervises	25,05	25,0	0,5 * RETRIEVE(sup	0,1 * INSERT(Employee)=178,18
works_on	27,59	27,59	0,43 * UPDATE(supervises)=110,86	
manages	22,24	22,24	0,5 * RETRIEVE(works_for)=4,37	
works_for	160,72	82,82	0,25 * INSERT(works_for)=164,58	
controls	25,05	43,94	164,26	4,37
			70,81	3,84

Figure 3: Visualizing database transaction costs and contents to the conceptual designer.

approach for conceptual schema tuning [19], we improve the conceptual schema by collapsing and subsequently dividing structures. By this, we (have to) restore the old structures, which the database design has originally modeled, but delete and newly generate relationship - to - entity references. According to this, assume the designer adds an optimization rule:

```

CSL> add conceptual optimization rule :
>   when component r2 r1 and CC(r2,r1) is (m,n)
>   and component r3 r1 and CC(r3,r1) is (p,q)
>   and m=p and n=q
>   and exists r4,r5,r6:
>     (component r2 r4
>      and component r5 r4 and CC(r5,r4) is (1,1)
>      and component r3 r6 and component r5 r6)
>   do
>     separate
>     (separate
>      (group
>       (group
>        (group (r4,r5) (1,1)) r2 (.,.)) r3 (.,.)) r4) r5
>      ;
>   Adding 1 new conceptual optimization rules.
    
```

Then, after applying the optimization rules we obtain a schema with lower operational complexities, easier implementation mapping and better behavior.

Figure 4 shows how RADD restructures the conceptual schema of Figure 1 according to the known population aspects, behavior specifications, evaluated performance properties, and the conceptual schema tuning rules. The dependency $Department \rightleftharpoons manages$, i.e. $CC(manages, Department) = (1,1)$ is maintained and visualized in the optimized schema by the path-key constraint K_2 , Figure 4.

Further rules for (internal) schema optimization can then be applied according to tuning techniques of the chosen DBMS, after a proper transformation has generated the logical schema. (See Section 4.1.)

Table 2: Design optimization rule - based on transaction costs.

highcomplexity(delete(R_1)) $\wedge$ highcomplexity(delete(R_2)) $\wedge$	entity(R_1) $\wedge$
	entity(R_2)
$((dcycle(R_1, R_3, R_2, R_4) \vee dcycle(R_1, R_4, R_2, R_3)) \wedge compatible(R_3, R_4)) \rightarrow separate(group(R_3, R_4, (.,.)), R_4)$

Figure 4: Optimized *Company* schema with path/ key constraint.

4 Generating the Database

The proper logical database schema is generated after the conceptual schema has been optimized, on basis of letting the database designer decide which *instantiated* transformation rules must be applied and which have to be ignored. This process is – as well as the review transformation, see [20] – a loop repeatedly applied to all structures and transformation rules, until no rule does fire anymore. An example for a transformation dialog is:

```
CSL> transform schema;
Transformation rule fires for 'group (Employee,works_on) (1,3)'.
Shall I do this transformation (RECOMMENDED) [Yn] ? y
Transforming ... done.
Transformation rule fires for 'group (Employee,works_for) (1,1)'.
Shall I do this transformation (NOT recommended) [yN] ?
Skipped.
Transformation rule fires for 'group (Project,controls) (1,1)'.
Shall I do this transformation (RECOMMENDED) [Yn] ? y
Transforming ... done.
Transformation rule fires for 'group (works_for,manages) (0,1)'.
Shall I do this transformation (RECOMMENDED) [Yn] ? y
Transforming ... done.
```

Here, the order in which the transformation rules were processed was determined in advance, by means of attaching precedences to the constraints. The constraint which is the more critical according integrity maintenance got the higher precedence. Then, immediately before the transformation, the constraint set was sorted with reference to this constraint precedence order.

The termination of the transformation process is given by the fact that the *Skipped* actions are stored as so-called *notactions* which do not fire again. In addition, the designer can set some options which, for example, influence:

- whether the transformation (and/ or conceptual schema optimization) is *interactive* or *non-interactive*;
- whether the type of transformation is *hierarchical*, *network*, *relational*, or *object-oriented*, that is, whether a hierarchical, network, relational or object-oriented model for the internal schema is used;
- whether and how attributes for internal keys are added to the output schema of the transformation (in case of a *relational transformation*);
- and, the size of some global variables indicating whether and how to collapse or separate internal structures.

4.1 Generating Database Structures and Constraints

The SQL (SQL-2) database schema definition code can be directly generated from the CSL interface:

```
CSL> export internal schema to SQL "Company.sql";
done.
```

Here, the SQL data definition code has been exported to the file "Company.sql". This file contains

- *Create table, primary-key, foreign-key, alter table, create index, create trigger and create procedure* statements, where the triggers and stored procedures are derived from the logically specified "behavior" properties and CSL functions.
- Advices for time of implementation of foreign-keys, even if they cannot be implemented directly (because of foreign-key cycles), and hints for the *enabling/disabling* of the foreign-keys and for the use of the stored procedures.

In this way, RADD generates on the base of a conceptual design (graphical design environment) and logical definition of procedures and database functions (CSL interface) a complete database implementation schema, that contains necessary and important hints for realization.

4.2 Advising the Generation of Form Applications

A well-known problem of most form generators and style guides for graphical database application development is that they restrict the programmer to a special implementation technique. These techniques are often not compatible to the logic of the implemented database schema. So, for instance [8, 9] only list examples, which are based on a database schema that is strongly hierarchical, tree-like organized. The implementation of an application is used to explain so-called *master-detail relations*, which are an isomorphic mapping of the database schema constraints. This is done without reference to concrete examples of data manipulation code, in form of SQL insert, delete, and update statements. This disallows many-to-many relationships of the implemented database (e.g., relations which have more than one foreign-key), and user masks which are logically organized as detail-master. It also does not enable cycles in the physical database schema, which some times can be straightforward obtained from a given conceptual database design, like the initially modeled schema of our example. (See also [20].)

Most database schemata which are found in style guide reference manuals should be considered as the exceptional cases for practical database applications, even for a database designed with a conceptual design tool. Well-designed conceptual database schemata and object models typically result in a range of integrity constraints to be implemented on the database, which contradict the advices of those manuals.

So what is wrong? We think that the recognition of such application designs must not be used to force redesign of the conceptual or logical database view. But, the application implementation should be guided by the tool which is used for the conceptual and logical database design as well. RADD has the capability to perform this task.

Consider the optimized conceptual and the transformed Company schema, that was exported to SQL. In our conceptual schema we had specified, that an *Employee* must work for exactly one *Department*. This enforces to create a master block *Department* and a detail block *Employee* in the application interface. In addition, the designer specified an ON-INSERT-CASCADE rule for the reference *works_for* $\rightarrow$ *Employee* and did also not decide to collapse *Employee* and *works_for*. This generates two problems of application development:

1. There is a many-to-many relationship path *Department-works_for-Department* on the SQL schema level.
2. If we have an application interface (a form) for *Employees*. and in this mask the user wishes to see at least the name of the *Department*, which the *Employee* is working for, then, in this form, *Employee* needs to be the master block and *Department* needs to be the detail block.

This confuses transaction management that is typically generated by a normal application generator or style guide, such that there is need to generate application trigger templates that already contain the full data manipulation code for such transactions. For instance for ‘delete of the current Employee’, the transaction generated from an application generator would first try to delete the *Department* record which is a detail block of the *works_for* record, which is a detail block of the *Employee* record. This results in a foreign-key violation iff there are still other Employees working for – or Projects managed by – this *Department*. Otherwise, it results in delete of the *Department* which may not be desirable. For this situation RADD can propose the following transaction:

```
--
-- BEGIN: disable Forms standard transaction handling here
--
--
-- DO NOT delete the record of the F__Department block
--
-- DELETE ALL detail records of F__works_for
-- INSERT CODE HERE:

-- DELETE from F__works_for
delete from works_for where ssn = F__Employee.ssn;
--
-- DELETE ALL OTHER detail records of F__Employee
-- which are instances of DETAIL BLOCKs OF THE DATABASE
-- INSERT CODE HERE:

-- DELETE from F__Employee
delete from Employee where sss = F__Employee.ssn;
-- COMMIT the transaction
commit;
--
-- END: re-enable Forms standard transaction handling
--
```

The example shows that the data manipulation code for such transactions is not easy. This maybe seen as reason for that current database application generators are not able to generate such code fragments – or leave the application programmer alone by forcing him to code the display and graphical change of all detail records. But, this may not be desirable, especially in cases of blocks which are containing only foreign-key attribute values of the master or internal keys. In our example, the *works_for* block is such a block.

Our work currently focuses on generation of templates, which are then containing complete data manipulation code examples for such user interfaces.

5 Conclusion and Further Work

RADD implements conceptual strategies for different user behavior on graphical design, a modular schema construction based on *units*, and incorporates into the conceptual design semantical data analysis and requirements specification. RADD analyzes design impacts for the later database maintenance. So, the database designer can chose his preferred strategy and can concentrate on conceptual design, since he is not required to refer to logical and physical design aspects of database implementation. Physical aspects which are inferred by the toolbox, are subsequently reflected to the conceptual design, such that the designer is notified for possible mistakes which have unadvantageous effect on the database implementation. Additionally, database transactions and transaction complexities are visualized to the designer in form of a conceptual representation. In sense of a conceptual schema to conceptual schema transformation, the design can be optimized automatically – according to transactions and transaction complexities which the designer agrees to be bottlenecks. The schema restructuring and optimization is based on a set of rewrite rules, which the experienced designer is able to extend. This makes database development more transparent, flexible and consistent.

The development of the RADD components

- *schema editor and graphical design environment* [4],
- *semantic acquisition* [1], and
- *behavior estimator/conceptual database optimizer* [20]

is already finished. Our mediate goals are (1.) to extend Csl modules (shown in Section 3.5) such that we also can support information hiding (*public*, *protected*, *private*) and inheritance, and (2.) to couple the RADD graphical database modules with Csl module specifications. We currently work on the automatical generation of *form* applications from Csl modules, which was presented in Section 4. In contrast to other application generators and style guides, this kind of automatical generation of application modules does not mean to give some proposal for the graphical layout of the application, but to generate the code which is necessary for button-callbacks of the database application interfaces.

In this way, RADD represents a *next-generation database design tool*, that guides the database designer through all steps of the database's development and animates the design for him with the goal to develop an efficient, complete, and consistent design.

References

- [1] M. Albrecht. Semantikakquisition im Reverse-Engineering. Technical Report I - 4/1994, TU Cottbus, Institut für Informatik, June 1994. (in German).
- [2] M. Albrecht, M. Altus, E. Buchholz, A. Düsterhöft, and B. Thalheim. The rapid application and database development (RADD) workbench – A comfortable database design tool. In J. Ivari, K. Lyytinen, and M. Rossi, editors, *Proceedings of the 7th Int. Conf. on Advanced Information System Engineering - CAiSE'95*, number 932 in LNCS, pages 326–340, Jyväskylä, Finland, June 14 - 16 1995. Springer-Verlag. <http://www.informatik.tu-cottbus.de/cs/dbis/caise95/caise95.html>.
- [3] M. Altus. RADD: Modeling of structure, semantics, and behavior of databases using HERM+. Technical Report Part 1: Data Dictionary & User Support, Cottbus Tech University, Computer Science, 1994. (Second intermediary report).
- [4] M. Altus. Meta-Modelling of Conceptual Data Base Design Using a Plausibility Function. In H. Kangassalo, J.F. Nilsson, H. Jaakkola, and S. Ohsuga, editors, *Information Modelling and Knowledge Bases VIII*, volume 38 of *Frontiers in Artificial Intelligence and Applications*, pages 113–129. IOS Press Ohmsha, 1997. 6th European-Japanese Seminar, Hornbaek, Denmark, May 28-31, 1996, <http://www.informatik.tu-cottbus.de/~altus/node4.html>.
- [5] C. Batini, S. Ceri, and S. Navathe. *Conceptual Database Design, An Entity-Relationship Approach*. The Benjamin/ Cummings Publishing Company, Inc., Redwood City, 1th edition, 1992.
- [6] J. Biskup. *Grundlagen von Informationssystemen*. Lehrbuch Informatik. Braunschweig, u.a., Vieweg, 1995.
- [7] P. Bunemann and Atsushi Ohori. Polymorphism and Type Inference in Database Programming. *ACM ToDS*, 21(1):30 – 76, March 1996.
- [8] Oracle Corp. *Oracle Developer/2000 – Forms 4.5 Advanced Techniques Manual*, 1996.
- [9] Oracle Corp. *Oracle Developer/2000 – Forms 4.5 Reference Manual Volume 1/2*, 1996.
- [10] P. Corrigan and M. Gurry. *ORACLE Performance Tuning*. O'Reilly & Associates, Inc., 1993.
- [11] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 6th edition, 1995.
- [12] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 2nd edition, 1994.

- [13] G.C. Everest. NIAM/OR Modeling: Student and Expert Solutions. In M. Albrecht and B. Thalheim, editors, *Proceedings of the Workshops Challenges of Application and Challenges of Design, at the ER'96 Conference*, Cottbus, Germany, October 1996.
- [14] J. Hartmann and L. Will. Werkzeuge zur Performanceanalyse von Datenbankanwendungen am Beispiel von ADABAS D. *Informatik Spektrum*, 20(4):225–228, August 1997.
- [15] J. Paredaens, P.D. Bra, M. Gyssens, and D.V. Gucht. *The Structure of the Relational Database Model*. Number 17 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin u.a., 1989.
- [16] A. Raihä and H. Manila. *The Design of Relational Databases*. Addison-Wesley, 1992.
- [17] D.E. Shasha. *Database Tuning – A Principled Approach*. Prentice Hall, 1992.
- [18] M. Steeg. From E-R to Optimal Database Schemes. Technical Report I - 7/1994, Cottbus Tech University, September 1994.
- [19] M. Steeg. CoDO – The Cottbus Conceptual Database Optimizer – and its Extensible Rule Model EC²A. Technical Report I - 2/1995, Cottbus Tech University, July 1995.
- [20] M. Steeg. The Conceptual Database Design Optimizer CoDO – Concepts, Implementation, Application. In *Conceptual Modeling – ER'96, 15th International Conference on Conceptual Modeling*, Cottbus (Germany), October 1996.
- [21] S.S. Su. Processing-Requirement Modeling and Its Application in Logical Database Design. In S.B. Yao, editor, *Principles of Database Design*, volume 1: Logical Organization, pages 151 –173, 1985.
- [22] Sybase - *Frequently Asked Questions, Section 8: Performance and Tuning*. http://reality.sgi.com/pablo/Sybase_FAQ/.
- [23] S.L. Tanimoto. *The Elements of Artificial Intelligence Using LISP*. Computer Science Press, 1987.
- [24] B. Thalheim. Database Design Strategies. In J. Paredaens and L. Tenenbaum, editor, *Advances in Database Systems, Implementations and Applications*, number 347 in CISM Courses and Lectures, International Centre for Mechanical Sciences. Springer-Verlag, Wien, New York, 1994.
- [25] B. Thalheim. Codesign von Struktur, Funktionen und Datenbanken. Technical Report I-05/1997, Computer Science Institute, Cottbus Technical University, 22. Februar 1997.
- [26] B. Thalheim. *Fundamentals of Entity-Relationship Modelling*. Springer-Verlag, 1997.
- [27] J.D. Ullman. *Principles of Databases and Knowledge-Base Systems, Volume I: Classical Database Systems*. Computer Science Press, 1988.
- [28] G. Vossen. *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme*. Addison-Wesley Verlag, Bonn, 1987.