RADD/raddstar

A Rule-based Database Schema Compiler, Evaluator, and Optimizer

Von der Fakultät Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr.rer.nat.)

genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Martin Steeg
geboren am 15. Februar 1965 in Weisel

Gutachter:	Prof. Dr. Bernhard Thalheim
Gutachter:	Prof. Dr. Dennis E. Shasha
Gutachter:	Prof. Dr. Hartmut Wedekind
Tag der mündlichen Prüfung:	20. Oktober 2000

RADD/raddstar

A Rule-based Database Schema Compiler, Evaluator, and Optimizer

Martin Steeg

Brandenburg University of Technology at Cottbus
Computer Science Institute

PO Box 101344
03013 Cottbus (Germany)

Technical Note. The document has been typeset using LaTeX$_{2e}$ and TeX$_{3.14159}$. The RADD/raddstar system described here was developed under Linux, kernel version 1.1.53, 2.0.34, 2.1.127, 2.2.7, and 2.2.13, on a 686 PC, and under Solaris 2.5.1 on a Sun Sparc. On both platforms, Linux and Solaris, X11 with the $\mathcal{K}$-Desktop environment (KDE) was used as window system. The diagrams and screendumps have been generated using xfig, xwd, xpr, xpaint, and kpaint. Some hacks were applied to the included Postscript files that were generated by xpr or xpaint, to scale and position the graphics appropriately.

Theoretical computer science has now undergone several decades of development. The "classical" topics of automata theory, formal languages, and computational complexity have become firmly established, and their importance to other theoretical work and to practice is widely recognized. Stimulated by technological advances, theoreticians have been rapidly expanding the areas under study, and the time delay between theoretical progress and its practical impact has been decreasing dramatically.

Michael Garey and Albert Meyer,
in the Series Foreword to
Foundations of Computing.

Contents

List of Figures

List of Tables

Abstract

Database design typically results in SQL create table commands and integrity constraints which are specified by foreign-keys, cascading deletes, etc. Specific products of the important database vendors provide a limited form of higher level design of the database relations and their associations. The design is then very close to physical design of the schema which is implemented under the associated DBMS. This design and specification approach has a rather limited scope and is difficult to handle. And, data profiles– such as the velocity, priority, and frequency of transactions, or the tuple numbers of relations –are also not considered by this traditional approach to database design and database application development. This way, operational behavior is not considered and can not be improved during database design.

Recent information system development approaches, starting with the design of the logical database schema by means of the database design tool of the DBMS vendor and then constructing the database applications with the help of a Web application builder, also do not consider these operational behavior details. Rather, they restrict the designer to construct the schema in a special way. This often is in contradiction with the actions which are necessary to optimize database performance.

In order to overcome the data modeling problems which are recognized at time of database maintenance, we developed an approach to database optimization at the conceptual level. We use an extension of the entity-relationship model, the Higher-order Entity-Relationship Model (HERM), and the workbench RADD developed for supporting HERM specifications. Although a high level of abstraction is provided in order to be user-friendly, in RADD data structures and integrity constraints can be specified together with data profiles, operations, and other application requirements.

In the thesis we present and discuss the RADD/raddstar system, which is the subsystem of RADD used to specify additional processing requirements, to evaluate and verify behavior properties, and to optimize the conceptual schema. We invest special interest on previous approaches to database modeling and database optimization, and define the data model that is used for RADD/raddstar's internal evaluations (RADD*). RADD* represents the items of the conceptual and internal schema by functional terms, and enables the user to add database application functions and behavior specifications to the graphical RADD database design. This way, we are able to analyse maintenance aspects of the designed schema and to find possible contradictions and performance bottlenecks. With or without the additional requirements that are specified by the database designer, the generation of schemata for implementation under consideration of, but independent from a specific DBMS can be used for the analysis of the given design, for the discussion of bottlenecks, and for the generation of design schemata with better operational behavior.

Part I

Database Design and Database Maintenance

In the real world an object simply exists, but within a programming language each object has a unique handle by which it can be uniquely referenced. The handle may be implemented in various ways, such as addresses, array index, or unique value of an attribute.

James Rumbaugh et al.,
in [RBP+91].

Chapter 1

Introduction

The thesis focusses on analysis and optimization of structural and operational dependencies during conceptual database schema design. The goal is to detect and solve problems which possibly appear when the database is running under a database management system (DBMS). The conceptual database optimizer that has been developed in this work analyses the correspondence between the HERM/RADD[1] conceptual database schema which is given or under design, and the operational performance and behavior of the database system that will be implemented based on that conceptual schema, using a DBMS.

1.1 Motivation and Overview

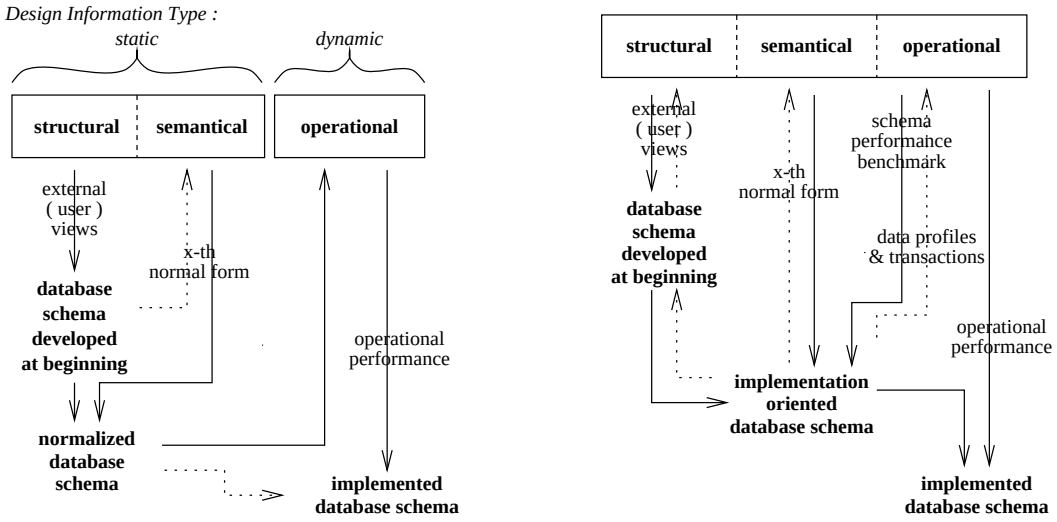
Traditional database development is based on waterfall approaches. The designer starts with requirement analysis, designs the conceptual schema, and translates it to the logical schema. Then, the logical schema is implemented using a special DBMS. This way, traditional data-driven approaches to information system development do not consider operational behavior in detail. Recent database design approaches, which start with designing the logical schema with the help of the design tool of the DBMS vendor, do also not consider these operational behavior details, but rather restrict the designer to construct the schema in a special way. This is often in contradiction with the actions which are necessary to optimize database performance.

In order to overcome the data modeling problems which are recognized at time of database maintenance, we developed an approach to database optimization at the conceptual level. We use an extension of the entity-relationship model, the Higher-order Entity-Relationship Model (HERM), and the Rapid Application and Database Development (RADD) workbench supporting HERM specifications. In RADD, data structures and integrity constraints can be specified together with data profiles, operations, and

[1] HERM/RADD entity-relationship schemata are used for illustration purposes in this work.

other processing requirements. In RADD, integrity constraints are not only considered as structural dependencies in form of inheritance and reference which are graphically specified (IS-A,REF), but also formally specified as cardinality constraint (CC), functional dependency (FD), inclusion dependency (ID), exclusion dependency (ED), afunctional dependency (AD), and path constraint (PREF,PCC,PFD,PID,PED,PAD).

Figure 1.1[a] and Figure 1.1[b] illustrate the differences between the traditional approach to database design and the RADD approach to database design.



[a] The Classical "Waterfall" Approach. [b] RADD's Data Profile oriented Approach.

Figure 1.1: Different Approaches to Database Development

In the thesis we present and discuss the RADD/raddstar system, which is the subsystem of RADD used to specify database processing requirements, to evaluate and verify behavior properties, and to optimize the conceptual schema. RADD/raddstar uses a high level of abstraction to allow user-friendly formal specifications, which can also be introduced in a comfortable way by means of a graphical user interface. (*RADD/raddstar GUI* respectively *RADD/raddstar Listener GUI.*)

1.2 Traditional Database Design

Traditionally. database design is a process of three major steps. The database designer starts with (1.) requirement acquisition, (2.) designs the conceptual schema, and (3.) developes internal data strutures with which the database is implemented. According to this, the architecture of databases has been commonly accepted as three-layered architecture, with external, conceptual, and internal view. The external view describes the application users's views to the database. These views are based on the appearing masks

to input new data or update existing data. Nowadays, such actions are performed by GUIs and Web browsers, like "Netscape" or "Internet Explorer", or Web applications which are implemented for special purposes. E.g., Java applets which are called from hypertext markup language (HTML) or extensible markup language (XML) documents and contain compiled code that is downloaded to the client, perform tasks such as retrieving and updating the database on the Web server. These Web tools allow to modify databases in a comfortable way.

Conceptual data models traditionally were considered as *the hierarchical data model, the network (data) model,* and *the relational model of data* (relational data model). These support modeling of "normal" data structures, i.e. structures that are of type traditionally used in computer applications, like *boolean, integer, float, string*, or combinations of these types *(records)*.

For the mid of the 70's implementation independent data models, the Entity-Relationship Model (ERM) [Che76], Nijssen's Information Analysis Method (NIAM) [Nij77, VB82], Functional Data Models (FDMs) [KP76, Shi81], and Semantic Data Models (SDM,IFO) [HM81, AH87] have been proposed. The latter models focus rather on *what kind of information* must be stored by the database than on *how to represent the information* by the computer. This way, they provide a larger degree of *data independence*. Therefore, the former "conceptual" data models (hierarchical, network, relational) are called "logical" data models today, while they represent an interface between the "new" conceptual models and the database implementation with the help of a DBMS.

Figure 1.2 presents the proceeding of the traditional database design approach using conceptual, logical, and DBMS data models. The conceptual data model is used for requirement analysis and conceptual modeling, such that it provides the external and conceptual views to the database. The logical data model is used to provide the conceptual view in the implementation language of the DBMS to the database, called logical view. And, the data model of the chosen DBMS provides the internal view to the database.

1.3 The Database Optimization Problem

Database optimization *(database tuning) is the task of making a database run more quickly* [Sha92]. There are different approaches to database optimization.

1.3.1 The Normalization Approach to Database Optimization

Using the *relational model of data* for database design [Cod70, Cod79], it has often been argued that *normalization* is a good approach to avoid *insertion*, *deletion,* and *update anomalies*, and so, to provide correct operational behavior of the database applications. But, although the result of the normalization process preserves *functional dependencies*, it

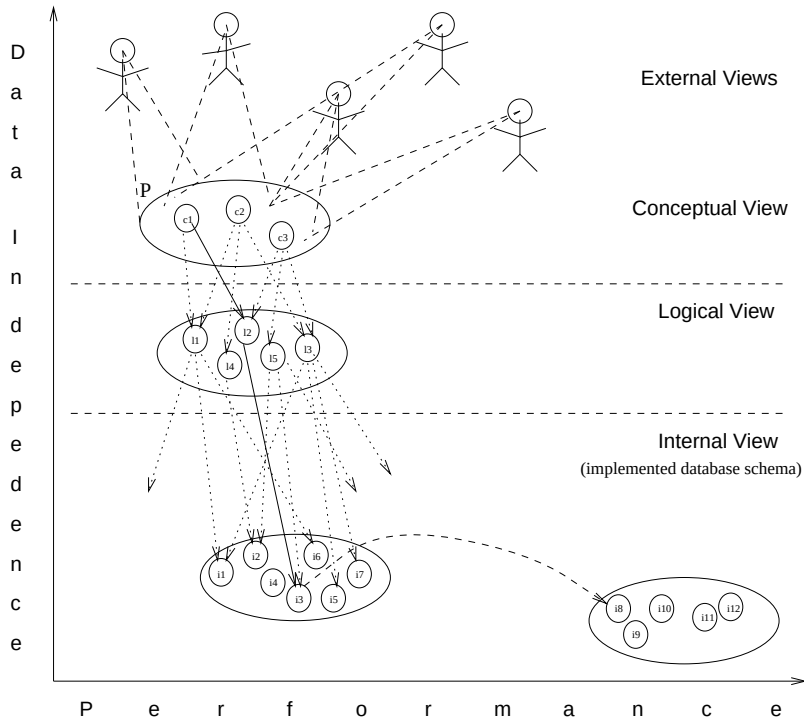


Figure 1.2: Traditional Database Design Approach: Requirements Analysis, Conceptual Design, Logical Design, Logical to Internal Schema Transformation, and Internal Schema Tuning.

was farreaching ignored that the relation schemata which are resulting from the normalization process define new *inclusion dependencies.* These must be additionally maintained, and, their maintenance is then due to applications, or their control (and possibly, *enforcement*) is implemented by database triggers and stored procedures. The normalization approach also requires that *joins* have to be applied to put the data into relation again, which are required by online applications or batch transactions, and which were previously stored by one relation and are spread over many relations after normalization.

Object-oriented database design approaches often give an impression that the implementation's data structures can be mapped closely to the real world data, and sometimes ignore that this may no longer preserve the integrity constraints which are underlying the data, or that it may not provide good operational behavior.

These actions, normalization and object-oriented structuring, employ the database management system with additional tasks and can make the whole database slower. The object-oriented policy of splitting application code into smaller parts– such that equivalent code fragments are used by different modules and programs although they are implemented only once *(reusage)* –sometimes conflicts with the goal of making the database application efficient.

1.3.2 The Transaction Runtime Tuning Approach

Transactions are sequences of select, insert, delete, and update operations either which are all successful and have effect to a new database state *(commit)*, or else, at least one of these operations fails such that none of the operations of that sequence has effect to a new database state *(abort* or *rollback)*.

Normally, the first time operational behavior and performance are looked at is when the database schema is implemented with the help of the chosen DBMS, the database is filled with a large amount of data, and the reaction time of the applied transactions is found to be not appropriate; i.e., it is not sufficient enough. Then, logical and physical (internal) database tuning actions which usually add indices to the internal schema, but may also restructure the physical structures and their connections completely, are applied.

1.3.3 The Rule-Triggering System Approach

Active database management systems (ADBMSs, see [DBB+88, CW94, AHW95]) which employ an event-condition-action (ECA) rule-triggering mechanism for transaction maintenance may be seen as an extension of relational and object-oriented DBMS technology. ADBMSs have been developed for application in computer-aided design (CAD) and computer-integrated manufacturing (CIM).

Rule-triggering systems (RTSs) are well-usable to decouple applications from database maintenance, since the DBMS can automatically perform maintenance tasks as soon as certain events occur. This allows to define additional constraints to the database which were normally implemented by applications, such that the applications need no longer to maintain the database's integrity. This way, although the internal database schema may be not completely normalized and applications do also not take care for protecting integrity constraints on special user actions, integrity constraints can be preserved. The important relational database management vendors of today have incorporated rule-triggering mechanisms into their systems, such that the database developer can specify the database schema and according integrity maintaining rules (triggers).

However, as demonstrated in [ST94a, SST94], the RTS approach must be used with caution, since a database schema with carelessly specified triggers can invoke transactions producing database states *which are far away from the desired result.*

1.3.4 The Web Application Design Approach

Nowadays, design approaches such that the database design tool provided by the DBMS vendor, e.g. Designer 2000 (Oracle) or SQL-Designer (Sybase), is used for conceptual, logical, and physical design, or object-oriented database design approaches [Car94] are

applied. These tools are used especially for the construction of today's Web and Web-based database applications.

However, although one could assume that these tools transparently explain the structural and operational dependencies to the designer, e.g. *foreign-keys, secondary indices* & *database triggers* and their relation to *select, insert, delete* & *update* operations, these tools are not only limited that they do support one special DBMS, but also that they require the database designer to construct the schema in a special way. This does not necessarily model the objects of the real-world as they are. For instance, the relational database design tools do not make transparent why they force the designer to construct the database schema in a hierarchical way.[2] This enforces that important database processing aspects are omitted during data design, since they can not be represented structurally using these database design tools. Also, the possibilities to specify integrity maintaining rules are not or only in a rather limited form available using these design tools (relational or post-relational).

The specification and verification of integrity maintaining rules is up to the database administrator (DBA) group, and, although claimed by a range of authors today there is no commonly accepted object-oriented data model ([ABD+89, Car94]), object-relational data model ([DD95]), or active database standard ([Con96]). There is also no general specification model for designing the new-generation database applications, nor is there any (commercial) design tool which allows to specify and analyse the requirements of these database applications, and to infer whether the design of the structures and the corresponding database triggers is good or not.

For the given reasons, the traditional database design approaches as well as the recent database design approaches are not sufficient. The recent design approaches with the help of the logical design tools of the DBMS vendors are not sufficient, since they do not overcome the performance problems which did appear after traditional database design. Beyond this, recent database structuring and database application structuring approaches do rather strongen than remedy database performance problems: Possible implementations of the schema are not considered and transactions are not acquired and not prototyped during first phases, since tools which generate and analyse transaction sequences in advance of database implementation (according the chosen DBMS) are rather rare. However, *most important transactions are often known in advance of system implementation and should be specified at an early stage* [EN89].

Since the physical tuning actions, which are necessary and are applied after logical database design with the help of the database command line interfaces by the DBA

[2] One good aspect of the invention of the relational data model [Cod70] was once seen in the property to not necessarilly construct hierarchical database schemata– the hierarchical data model and the network data model did require the database schema to be necessarily hierarchical.

group, do not adapt the conceptual and logical representation of the database and do not change the external views of the database users, a misbalance between the database's internal representation and the database's external representation often appears after tuning. Therefore, whenever designing databases for practical applications it is desirable to find already during design acceptable compromises between necessary maintenance of integrity constraints and structuring of database application code, and efficient operational behavior on the other hand.

1.4 Conceptual Database Design Optimization

In this work, we present an approach to extend database design such that operational behavior and performance can already be looked at during conceptual design time. This requires that details for transformation from the conceptual schema to the internal schema which is used for implementation with the help of a special DBMS and the implementation's behavior must already be considered during conceptual design. However, the person who is performing the conceptual database design must not be confronted with details of the database implementation.
So, *conceptual database design optimization* can be done the following way:

1. The system (the conceptual database design optimizer) automates the database design transformations, and, different transformation kinds must be used such that the different results are compared and the best transformation is chosen.
2. The system supports the evaluation of the fitness of the operations and transactions that are identified on the internal schema, and reflects the fitness of the internal operations to fitness of *conceptual operations* which are presented to the designer.
3. The system enables schema restructuring according the bottlenecks which the database designer agrees to, such that it is possible to show the database designer alternative design representations of the *mini world* that is considered.
4. But, the system has to hide the data processed internally (because they may not be understandable to the database designer), and reason in the database designer's language why modifications of the schema are proposed or made automatically.

1.4.1 Conceptual Database Optimization Aspects

Population aspects of the internal database influence the response time of database operations drastically. We have therefore to consider these population aspects. These include the uniformity of data, tuple numbers (numbers of tuples) or correlations between them, and criteria whether certain sets of the database are changed frequently or not.

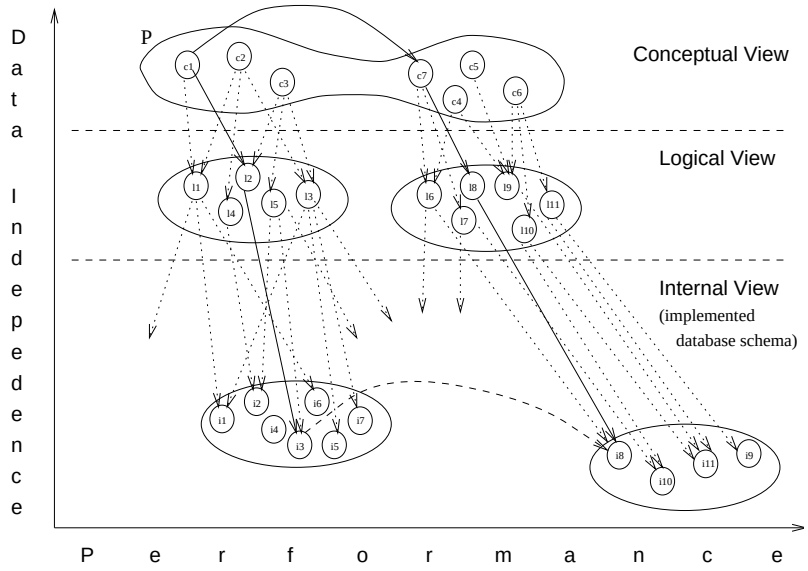


Figure 1.3: Enhanced Database Design Approach: Conceptual Design, Conceptual Tuning, Logical Design, Logical to Internal Schema Transformation (Internal Schema Tuning).

Implementation aspects to be considered by the conceptual database design optimizer are anomalies, referential dependencies, costs of join operations, and triggering actions. These criteria, population and implementation aspects, are typically used for physical design and restructuring decisions only, because these criteria are too early omitted in "normal" database design approaches. But, most often they can already be inferred during database design. The aim of the current work is not to adapt the objects of the mini world to the conceptual, logical, or internal database schema, but to adapt the database schema to the objects of the real world, i.e. to discuss the semantics of the objects and the problems which are detected according the behavior of transactions with the conceptual designer.

1.4.2 Principles of Conceptual Schema Optimization

Figure 1.3 illustrates the proceeding of conceptual database design optimization:

> *The conceptual database schema c_1 is inspected for its most obvious implementation schema (internal schema). The implementation schema i_3 is inspected for its bottlenecks, and how it is possible to tune (optimize) the database system.*

From this internal design information we reflect to the conceptual design as follows:

1. *Bottlenecks detected on the internal schema are located on the conceptual schema.*
2. *The conceptual designer is informed on bottlenecks of the given schema in a rather informal way, i.e. by using terms that are presented in the conceptual schema. A modified conceptual schema c_7 is generated and proposed to the conceptual designer.*

Whenever the designer accepts the schema that is proposed by the system– or considers the proposed schema and eliminates the bottlenecks of her/his conceptual schema –, then it is possible to derive the logical schema l_6 or l_8 and to generate the more optimal internal schema i_8 directly, on the basis of the conceptual database schema.

1.4.3 Physical Schema can still be optimized further

Several problems of schema tuning (optimization) can not be solved only during conceptual design, and by logical design neither. Bottlenecks are sometimes DBMS specific, such that there can not be given general arguments for a designed schema's "well-fitness". But, as Figure 1.3 (and the introducing conceptual database optimization scenario at the beginning of Chapter 5) illustrate, certain bottlenecks can be omitted by inspecting the implementation schema that can be derived in a straight-forward manner from the given conceptual schema. In this way, the task of database schema optimization can– at least partially –be performed during conceptual design. This adds a new dimension of transparency to the whole database design process– since optimization can be moved to earlier design phases where, consequently, the optimization aspects can be discussed with the database designer, and not only with the DBA group.

1.5 Related Work

Su examines in [Su85] data profiles and their relation to schema design. *Wiederhold* [Wie87] works out exhaustive considerations on database operation complexities. Also, [CDKK85] give in the *The Design and Implementation of the Wisconsin Storage System* a good examination and argumentation for the design issues of a general applicable and extensible database storage manager. *Korth and Silberschatz* present in [KS91] considerations on benefits and drawbacks of different physical data organizations and their

appropriaty for certain applications. *Shasha* covers in [Sha92] the most *principled proceeding to database tuning.* He uses different *scenarios* and brings many aspects together, which were not considered by previous approaches to database tuning. *Dunham* [Dun98] gives in the *handbook* an exhaustive documentation on *operating system and database performance tuning.*

Details of schema fitness evaluation and possible conceptual to internal transformations can be found at different places. *Hainaut* gives an evaluation framework for schema fitnesses based on a cost function approach [Hai89]. *Halpin* [Hal90, Hal91] is the first one using the term *Conceptual schema optimization. Campbell* [Cam94] considers "anchors" that are found in the information structures and are not changed, even if the database schema will be tuned. *Van Bommel* [BWL94, Bom94] shows how different internal representations of a conceptual schema can be derived such that it is possible to choose the "best". Van Bommel uses a *data profile* approach based on tuple numbers. The doctoral thesis [Bom95] summarizes *van Bommel's* approach.

Comparison of the RADD/raddstar Approach to the other Database Optimization Approaches.

The latter mentioned authors, *Hainaut, Halpin, Campbell,* and *van Bommel* and their co-workers use schema mutations which derive optimal implementation schemata from a given conceptual schema. I.e., the term *optimization* of these works must rather be looked at what we here denote as *transformation.* Furthermore, although the cost functions given in these works seem to state plausible and well-designed estimations, they omit– like several normalization proposals did before –dependencies of database operations, which are result of references and cardinality constraints, for instance.

The purpose of this document is another, i.e. not to give a cost function approach for conceptual database design only, but also to relate to what's going on in a practical database environment. Therefore, we use a *hybrid version* of schema valuation and optimization that combines both, conceptual database representations and physical (internal) cost estimations. To present the internal costs to the conceptual designer we use a conceptual schema to internal schema mapping that uses references of the internal structures to the conceptual structures from which they were derived, called *preceders.* By means of the preceder mapping, the contents and costs of conceptual transactions are evaluated and bottlenecks are marked on the conceptual schema.

In contrast to the other approaches to logical and internal database optimization, such as [Sha92] and [Dun98], the approach presented here intends to optimize the conceptual view to the database, and not its logical or physical representation. The approach given here constructs a conceptual schema from a conceptual schema, such that the new schema has the properties of the conceptual data model that was used to construct the given

conceptual schema. Additionally, the internal schemata which are used to evaluate the fitness of the conceptual schema, to detect bottlenecks on that schema, and to gather criteria for better conceptual schema design, can be shown to the database designer and exported to the data definition language of the DBMS which is used to implement the database. We generate a special form of SQL-92 create table, index, view, and procedure definition statements, that is, as we expect, also well usable for the forthcoming SQL database standard.

1.6 Outline of the Thesis

The thesis is structured into three parts:

Part I describes database design methologies and strategies to model databases, and presents the database design strategy of the RADD workbench.

Part II describes the underlying theoretical concepts and the implementation of the *RADD/raddstar Conceptual Database Design Optimizer*.

Part III presents an application scenario of the RADD/raddstar Conceptual Database Design Optimizer and gives concluding remarks on the approach.

Figure 1.4 presents a road map to read the thesis.

The reader who wants to quickly read the thesis, may follow the arrow labeled "q" and skip Chapter 2 to 8. The reader who is interested in the results of schema evaluation, transformation, and optimization, may follow the arrow labeled "o" and continue with Chapter 8, which contains the application scenario of the RADD/raddstar Conceptual Database Design Optimizer.

Chapter 2 presents the data models and database management system types which were traditionally used and which we have mentioned in this beginning Chapter. Chapter 3 continues, by presenting data models including advantageous data representation concepts. Chapter 2 and 3 give an impression on some DBMS implementation concepts as well. The reader may skip certain Sections of Chapter 2 and Chapter 3, and, for instance, read only about relational databases and the RADD database design model. Hence, the reader follows the arrow labeled "d", and continues with Chapter 8 which contains the application scenario.

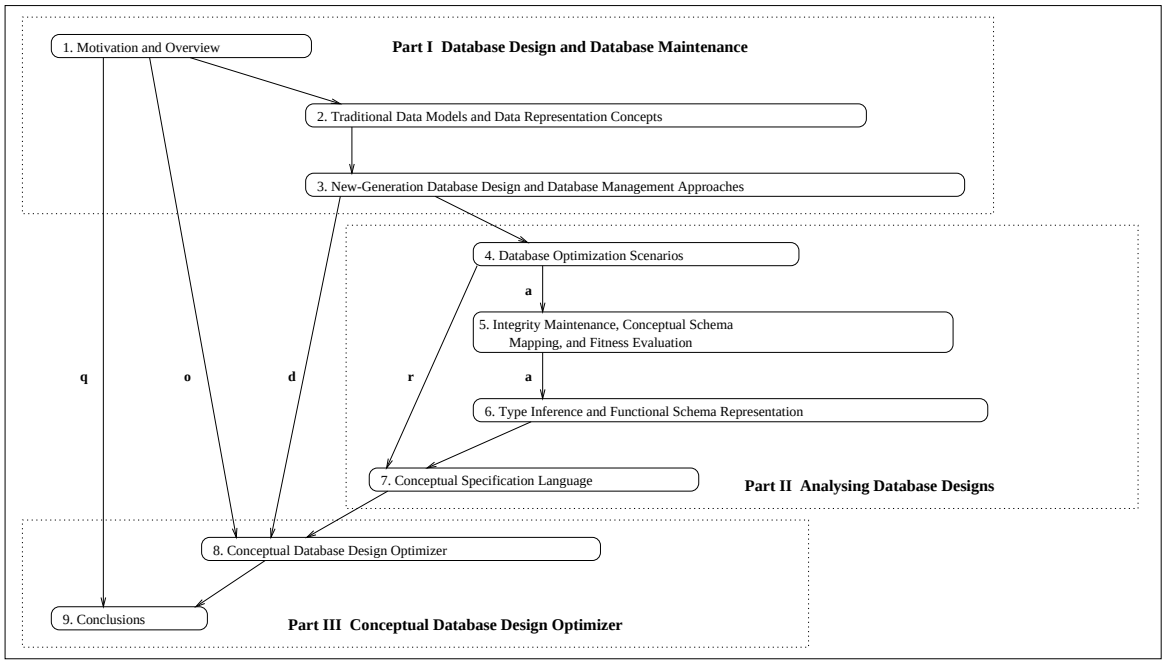


Figure 1.4: Road Map to read the Thesis.

The reader who is interested in database optimization scenarios, in the conceptual modeling concepts of RADD and in the conceptual design specification language provided by the RADD/raddstar system should follow the arrow labeled "r", and the one who also wants to know about the implementation of the RADD/raddstar system should follow the arrow labeled "a" and read the whole thesis.

Chapter 2

Traditional Data Models and Data Representation Concepts

A data model is a group of concepts for specifying a database, that has two parts ([Ull88a]):

1. A notation for describing data, and
2. A set of operations used to manipulate that data.

A database management system (DBMS) is a collection of programs to create and maintain a database. These programs are used as a general purpose software system for specifying, constructing, and maintaining a database for various applications. In comparison to file management systems, advantages of DBMSs are the integration of data and application programs, the support for multiple user views, and the description of the database's structure by the database itself (by means of special tables storing information about all attributes, tables, indices, view, and so forth, called "catalog" or "data dictionary").

In Chapter 2 we present data models that were traditionally used for modeling the section of the real world for which the database is needed, called mini world. After requirements collection, the mini world is described by a conceptual schema. Section 2.1 gives an overview on data modeling concepts by means of a conceptual schema, introducing the entity-relationship model of Chen [Che76]. For illustration purpose, this Section uses a *Company Schema,* which is a modified version of that found in Elmasri and Navathe [EN89], and, from which we will use a part to illustrate conceptual database design optimization in Chapter 4.2. The *Company Schema* used in the current Chapter will be extended and adapted by representation concepts of the other data models in the following Sections. The hierarchical data model and the network model (Section 2.2) are described next. Then, we present the relational data model (Section 2.3). In Section 2.2 and 2.3 we also consider some implementation concepts of hierarchical, network, and relational DBMSs. Section 2.4 summarizes the considerations of the Chapter and gives an outlook

on the data representation concepts that are used by new-generation database systems, which will be considered in Chapter 3.

2.1 The Entity-Relationship Model

In contrast to the data models which were traditionally used for database design, the entity-relationship model (ERM) focusses rather on what kind of information is to be stored, from the conceptual viewpoint, than on how to represent it by the computer. It is closer to the users' perceptions such that it forms a language for requirements acquisition– which is done as the first step of the database design process and constructs the database model from the external users' views. The hierarchical data model, the network model, and the relational data model, which preceded the ERM as conceptual data models, are today rather looked at as data models for providing the result of conceptual schema to physical schema mapping, representing the conceptual schema in the implementation language of the DBMS.

A diagram of the ERM models the mini world based on the following concepts:

1. entity types, which model objects that are living independently from others in the mini world, respectively; in the entity-relationship (ER) diagram entity types are represented by boxes;

2. relationship types, which model objects that are not living independently from others in the mini world and are used to model associations between the entity objects; in the ER diagram relationship types are represented by vertices;

3. attributes, which describe properties of the entity and relationship objects; in the ER diagram attributes are represented by circles;

4. and multiplicities, on the relationship types, which describe how much objects of the first entity type have connection to objects of the second entity type; in the ER diagram multiplicities are drawn near the connecting lines between entity and relationship types.

Example. (Company Schema)

1. A company is organized into departments. Each department may have several locations, and it has a name, a number and an employee who manages the department. We keep track of the start date when that employee started managing the department.

2. A department has a number of projects, each of which has a name, a number, a single location, a project start date, and a duration.

3. We store each employee's firstname, lastname, title, social security number (ssn), address, sex, and salary. An employee works for one department and has skills according which he his scheduled on projects. He works on several projects, which are not necessarily controlled by the department he is working for. We keep track on the number of hours per week that an employee works on each project.

This mini world is represented by the ER diagram (schema) in Figure 2.1.

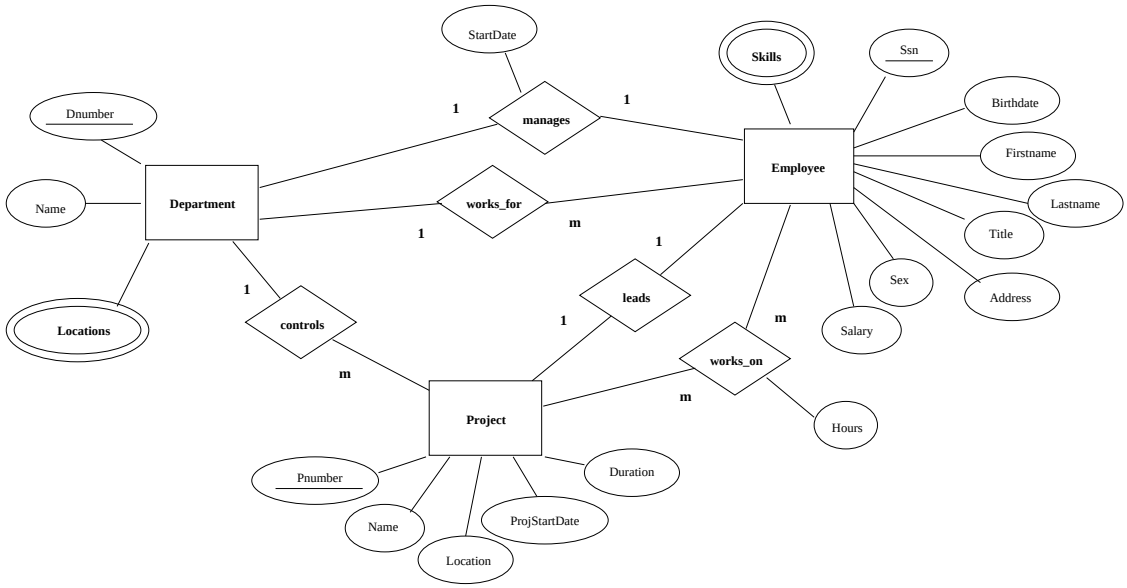


Figure 2.1: Entity-Relationship Schema for the Company Database.

The attributes *Dnumber*, *Pnumber* and *Ssn* can be used to determine the entities *Department*, *Project*, and *Employee* uniquely. Therefore, they are underlined in Figure 2.1. We also call them *key attributes*, or simply *keys*. The attributes named *Locations* and *Skills*, on the other hand, can be *multivalued*. Therefore, they are drawn using double-lined circles.

For the relationship types of this schema, we have not drawn key attributes. In case of many-to-many ($m : m$) relationship types, the instances of the relationship types are normally uniquely identified by the combination of the according entities' key attributes. For example, *works_on* records are determined by the values of the $Employee.Ssn$ and $Project.Pnumber$. The relationship types which are $1 : m$ need only the key attributes of the many (m) side to provide uniqueness of their instances. For the relationship types which are $1 : 1$, we choose one of the entity types, from which the relationship type then gets its key attributes. E.g. for the *manages* relationship type, $\{Ssn\}$ as

well as $\{Dnumber\}$ could be chosen as keys, such that they are *candidate keys*. However, since *manages* models the property how a *Department* sees its manager (which is an *Employee*), it is more natural to assign manages the key of *Department*, that is $\{Dnumber\}$.

So, for this example the key mapping for the physical design level can be given as follows:

- The physical *Department* type gets the key $\{Department.Dnumber\}$,
- the physical *Employee* type gets the key $\{Employee.Ssn\}$,
- the physical *Project* type gets the key $\{Project.Pnumber\}$,
- the physical $works\_for$ type gets the key $\{Employee.Ssn\}$,
- the physical *manages* type gets the key $\{Department.Dnumber\}$,
- the physical $works\_on$ type gets the key $\{Employee.Ssn, Project.Pnumber\}$,
- the physical *leads* type gets the key $\{Project.Name\}$,
- and the physical *controls* type gets the key $\{Project.Name\}$.

However, an entity type is not restricted to have only one key attribute, but this is the most frequent case. It it also possible that a many-to-many relationship type gets additional key attributes– besides the key attributes which it inherits from the entity types. Assume, an *Employee works on* a *Project* 3 hours on one weekday and 6 hours on another weekday, and we wanted to keep track not only on the total time which he works on that project per week, but also on the individual time according the different weekdays. Then we had to add an attribute– e.g. $Weekday$ –to the $works\_on$ relationship type, such that on the physical level the key of $works\_on$ were $\{Employee.Ssn, Project.Pnumber, works\_on.Weekday\}$. This consideration could be continued, e.g. by presupposing that the employee works on the same project and same weekday more than one period of time, and so forth. However, to keep the schema simple here, we assume that the company only keeps track on the hours per week which an employee works on a project.

Introducing artificial keys during conceptual database design. If an entity-relationship schema is transformed to a physical database schema, then most of the entity types which have not already an integer-typed key get an additional integer-typed key, such as $Employee.ID$. Such *artificial keys* are frequently used in database realizations for the following reasons:

1. They can be used to substitute *multi-column* keys or *unique indices* on more than one column, which generally are expensive in maintenance.

2. And, they provide faster query evaluation, especially if the key of the physical database structure would otherwise have a long attribute (which is character-typed a.s.o.), or has three, four or more attributes.

On the other hand, artifical keys additionally require the generation of unique key values, and, if there is an attribute which uniquely determines the objects of the structure, like $Employee.Ssn$, and there is an additional artifical key, like $Employee.ID$, then both attributes must be kept uniquely all the time. (For reason that the artifical key does not automatically provide that the real key is a key anymore!)

Therefore, the database design tools which advise to conceptually model ID– or *number* –attributes, such as *ErWIN* or the design tools of the relational DBMS vendors *(e.g. Oracle, Sybase, or Informix)*, are not to be considered as conceptual database design tools– from the author's viewpoint – since they already introduce implementation details into the high-level design of the database.

2.2 The Hierarchical Model and the Network Model

This Section surveys on the hierarchical and the network data model, and gives in Section 2.2.6 an example of coding the Company Schema using a network DBMS.

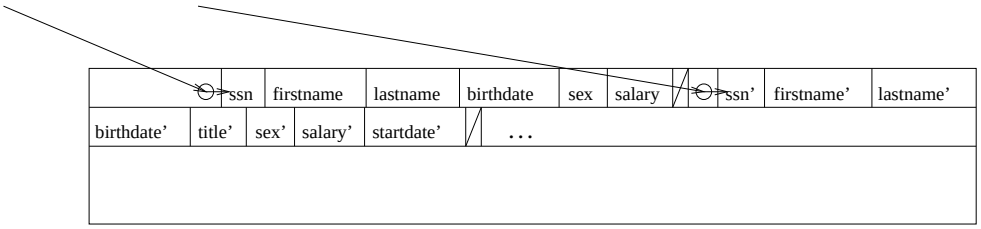


Figure 2.2: Organization of Records in a Hierarchical Database.

2.2.1 The Hierarchical Data Model (Hierarchical Model)

The hierarchical data model it the oldest of the conceptual data models and supports differently structured data of variable length, which are physically stored in heaps. For illustration let us assume that an employee $e1$ works for a department and another employee $e2$ manages the department, such that $e1$ has attributes Ssn, $Firstname$, $Lastname$, $Birthdate$, $Address$, Sex, and $Salary$, and $e2$ has the additional properties $Title$ and $StartDate$. This can be physically represented like shown in Figure 2.2.

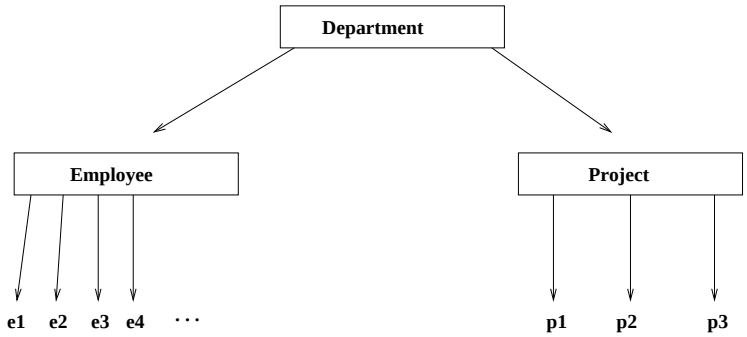


Figure 2.3: Logical Tree Organization of Record Types in Hierarchical Databases.

2.2.2 Integrity Constraints in Hierarchical Databases

The hierarchical organization of data records is intuitive to human thinking which structures the world in a hierarchical tree-organized manner, using roots which do have subnodes which continously do have subnodes again, until the leaves are reached. See Figure 2.3. The tree-organized structuring implies that inheritance constraints are built in each schema. These are:

1. Each record except a root record can not exist without having a parent record, such that records of the child record sets must ever be connected to a parent record, and the parent record can not be deleted from the parent record set as long as children records exists that are connected to it.

2. And, if a record logically has more than one parent (like the instances of many-to-many relationship types in the ERM), the record must be duplicated for each parent.

Therefore, a drawback of the internal tree structure of hierarchical databases is the problem of redundancy: using tree storage the same data may be represented twice or more. Consider the schema shown in Figure 2.3 and let us assume that the employee $e2$ works on project $p1$, such that– naturally –the $p1$ sees $e2$ as one its participants. This latter association could be understood as a function (or relationship type) *has_participants* from *Project* to *Employee*.

To represent this kind of association between child records of different paths in the tree, the hierarchical model makes use of records which form the association only. Consider the hierarchical schema in Figure 2.4. Here we have two records modeling the association between $e2$ and $p1$, the EwP record and the PhE record. Therefore, a better way is to model (and to implement) one record (EwP') which gives the association, and another, so-called *virtual parent-child record* ($VPCR\ PhE'$), which is a pointer to the association

record from the other path of the tree. In Figure 2.4 we also see another use of VPCRs. That is the *Manager* record, which represents the special manager role of the Employee $e4$, and, which has a pointer to the root of the tree. Using such records, it is possible to bypass the hierarchy in hierarchical databases.

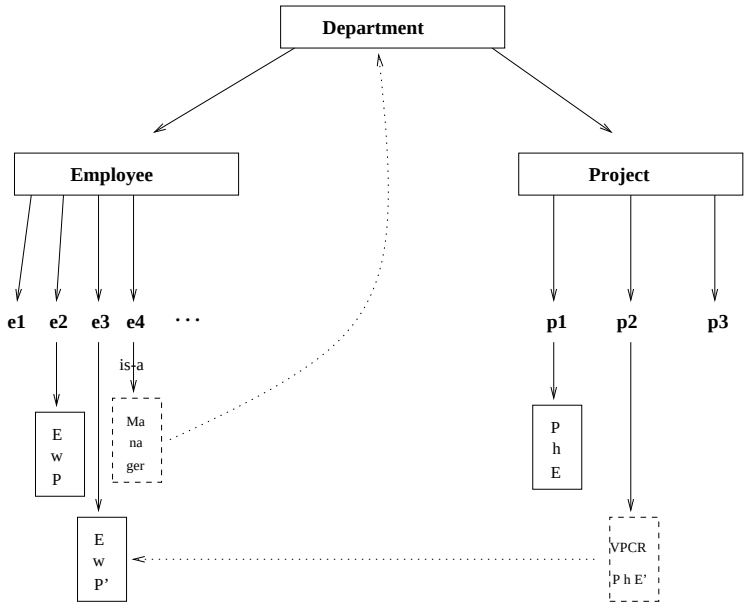


Figure 2.4: Associations between Record-Types, and Virtual Parent-Child Records.

2.2.3 Hierarchical DBMSs

Illustrative examples of hierarchical database applications, and of concepts to implement databases by use of IBM's hierarchical DBMS *IMS* are given in [KL78]. Hierarchical DBMSs make use of physical data storage techniques which apply additional indices to the records that are putted in heaps. For example, $HISAM$ (Hierarchical Index Sequential Access Method) is such a storage method. The storage techniques of the IMS system are discussed in [KL78] as well.

Chapter 11 of [EN94] explains which way the *Company* schema is implemented using IMS. The record types which are used by the hierarchical DBMSs are similar those which we will present by the network database implementation code in Figure 2.8. We omit therefore the code representation for the hierarhical implementation of the Company Schema here.

2.2.4 The Network Data Model (Network Model)

Comparing the entity-relationship model (Section 2.1) with the hierarchical data model (Section 2.2.1) one can recognize a discrepancy between these methodologies:

- The ERM uses a notation, which is based on entity and relationship types representing the entity sets (called entity occurence sets) and the relationship sets (called relationship occurence sets); as we will see in the forthcoming Sections, this enables us to specify the semantics of the ERM in a fashion of logic formulae, based on equations and predicates.

- The hierarchical data model bases on directed graphs which are specifycing rather the programmatical issues to implement and traverse (or navigate) the database, from the viewpoint of the operations which have to perform these tasks. For reason of the directed connections, associations which are many-to-many, like the $m : m$ relationship type *works_on* in Figure 2.1, are not directly supported by the hierarchical model, but must be represented from both viewpoints, the one of the *Employee* and the one of the *Project*, which requires to add a separate record-type in the hierarchical representation (PhE respectively PhE').

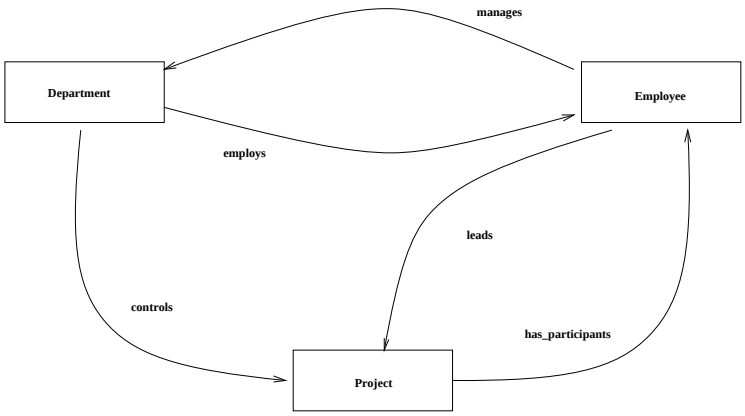


Figure 2.5: Bachmann Diagram (Network) for the Company Database.

We are now introducing the *network data model* which is based on directed graphs (networks) as well. A familiar representation of network schemata is given by *Bachmann* diagrams. Figure 2.5 shows the Bachmann diagram according the ER-Schema in Figure 2.1. Here, the relationship type that was named *works_for* in Figure 2.1 is named *employs*, and the relationship type *works_on* has now the name *has_participants*.

In comparison to the arrows in hierarchical schemata which have the meaning of pointer of a record to another record (one-to-one mapping), the arrows in the Bach-

mann diagrams specifiy an association between a record– called owner –and a set of records– called members–, which is a one-to-many mapping. For instance, the arrow from *Department* to *Employee* which is labeled *employs* in Figures 2.5 specifies a one-to-many owner-member relationship between *Department* and *Employee*. To provide better understanding of owner-member relationships we use in the sequel relationships which have one angle on the owner-side and two angles on the member-side ($\longleftarrow\!\!\!\!\!\rightarrow$).

There is a difference between the logical network schema and the physical network schema. Figure 2.5 contains two mutual owner-member relationships, which are *employs* and *manages*, and *has_participants* and *supervises*, respectively.

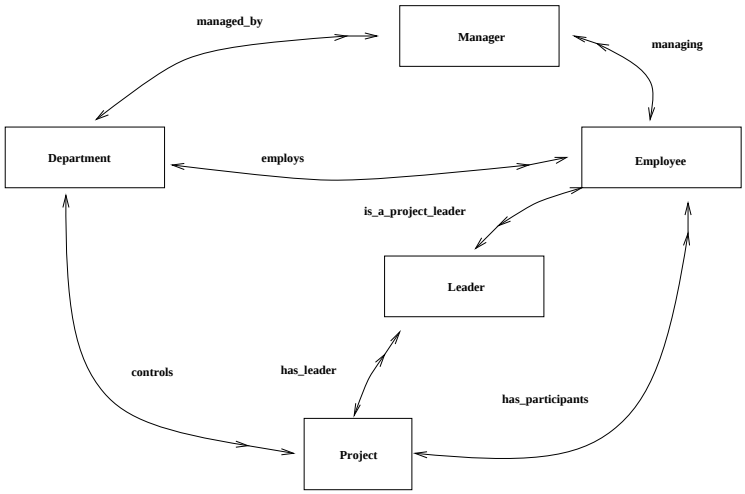


Figure 2.6: Physical Network Schema for the Company Database.

These are considered as many-to-many relationships by the network implementation, such that additional *logical record-types* (record-type is analogous to entity-type of the ERM) must be added. These additional structures are called *kett-entities*.

Assume *Manager* and *Supervisor* are the kett-entities that are added to the schema in Figure 2.5, then Figure 2.6 shows how the physical network schema for the Company database looks like. This way, the latter schema specifies additional *owner-member sets* (one-to-many relationship types) which are *Department-Manager* (*managed_by*), *Employee-Manager* (*managing*), *Project-Leader* (*has_leader*), and *Employee-Leader* (*is_a_project_leader*). The other owner-member sets, *Department-Employee* (*employs*), *Project-Employee* (*has_participants*), and *Department-Project* (*controls*) were already part of the conceptual representation in Figure 2.5.

The hierarchical model and the network model are similar, considering their concepts. E.g. they have in common that they specify the database according the navigational semantics of their operations. Additionally, they have a property making them *object-*

oriented in some sense [Ull88a], namly their physical handling of records by addresses. These can be looked at close to the representation of object identification mechanisms in object-based database models (refer to Section 3.1.3 and 3.2).

2.2.5 Integrity Maintenance in Network Databases

The network database implementation makes use of so-called *set-insertion* and *set-retention* options (integrity constraints) which are specified for the owner-member sets. Let us illustrate these options refering to the physical network schema shown in Figure 2.6.

Set-Insertion Options.

1. AUTOMATIC. Whenever a new record is added to the member set then it is added automatically to the owner-member set as well; for instance, if *automatic* is specified according the owner-member relationship *employs* and its *Employee* member-record sets, then a new record which is inserted into the *Employee* set, is also connected to some member set of the *employs* relationship. The member-record set *(current record set)*, which the new *Employee* is connected to, is determined by the current owner of the *employs* set, that is the current record in the *Department* record set.[1]
2. MANUAL. A new record which is added to the member set is connected explicitely to one of the owner-member sets; for instance, if *manual* is specified according the owner-member relationship *managed_by* and its *Manager* member-record sets, then a new record which is inserted into the *Manager* set will be connected manually by the application to some member set of the *managed_by* relationship.

Set-Retention Options.

1. MANDATORY. The owner-record of the owner-member set can not be deleted if there is still any member in the member set for which it is the owner; for instance, if *mandatory* is specified according the owner-member relationship *employs* and its *Employee* member-records, then an *employs* record can be deleted only after all records of the *employs* member set which have the *Department* as owner are either deleted or connected to another member set of the *employs* relationship.
2. FIXED. Like in *mandatory*, a record can not live independently from an owner. Moreover, once a member record is inserted into an owner-member set, it is *fixed*. That is, it can not be re-connected to another member set. Consider the owner-member relationship *managed_by* of *Department* and *Manager* and assume it is

[1] The resposibilty to set the current record set correctly is due to the programmer.

specified *fixed* according *Manager*; then, a manager represented by a record in the *Manager* set which manages not anymore the *Department*, must be deleted, but can not be changed to become the manager of another *Department*.

3. OPTIONAL. An owner-record may be not an owner or a member-record may be not connected to any member set of the owner-member relationship, respectively; assume a *Project* may not be controlled anymore by a *Department*, e.g. because that controlling *Department* was closed. Then the set-retention option of the *Project* member set of the *controls* relationship is specified *optional*, such that a *Project* can be made independent from any *Department*.

Useful Combinations of Insertion and Retention Options. Although not forbidden by the Codasyl specifications, not all combinations of insertion and rentention options make sense. Assume insertion is *automatic* for the *Employee* member records of the *employs* owner-member relationship (between *Department* and *Employee*); that is, an *Employee* member record is forced to be connected to an *employs* set. Then, it may not be wishful to use retention *optional*, since this would make it possible to disconnect *Employee*s from their *employs* set, as if insertion were *manual*. Accordingly, the insertion/retention options *manual-mandatory* and *manual-fixed* should be used neither, such that we propose to use only the following combinations of insertion and retention options: *automatic-mandatory*, *automatic-fixed*, and *manual-optional*.

Record Positioning in Sets. The records in the member sets of owner-member relationships can be ordered ascending or descending by fields, such that whenever a new record is inserted into the member set, it is positioned at the according place. In his coding, the programmer can also use *first*, *last*, *next*, and *prior* commands to position records at the according places in the sets.

2.2.6 Implementing the Network Database

In Section 2.2.4 we have shown only the names of the record types of the network database schema. In Figure 2.7 we see the more concrete structuring of the record types *Department*, *Employee*, and *Project*.

The schema in Figure 2.7 includes that a *Department* can supervise other *Departments*, and reversally, a *Department* can be supervised by another *Department*. But, we have not used separate record types for the different roles of the supervises relationship such that it is represented by the record type *SupervisingDepartment* only. Here, we use the

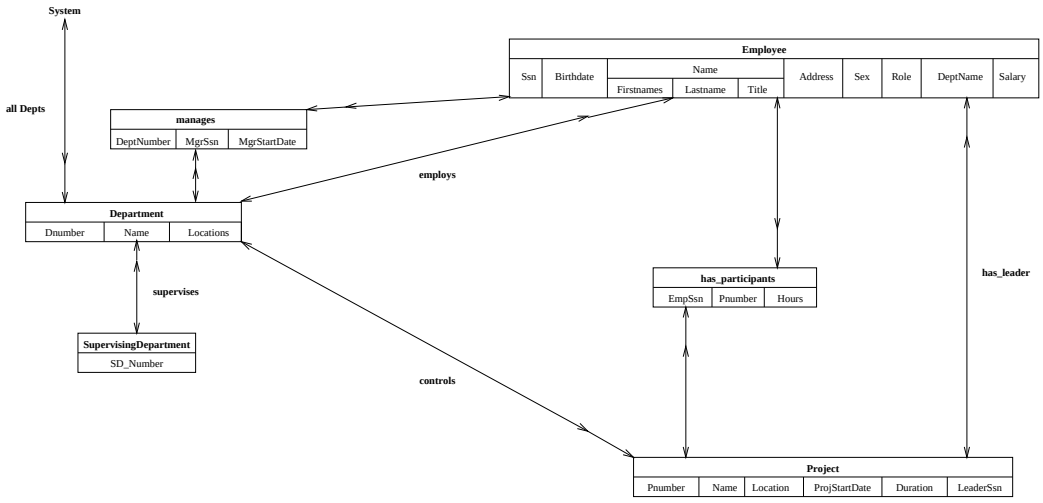


Figure 2.7: The Network Schema for Implementation of the Company Database.

foreign-key key attribute SD_Number representing the *supervised_by* role of that association, which is a valid option. That a SD_Number in the $SupervisingDepartment$ record is also a $Dnumber$ in a $Department$ record can be ensured either by the specification of the network database schema or by the applications which are used to modify the database. This way, the relationship type *supervised_by* is not specified by an owner-member set in Figure 2.8. The same way we must take care that a $DeptName$ in the $Employee$ record is really a $Name$ in a $Department$ record. The $EmpSsn$ and $Pnumber$ could be assumed to be foreign-keys as well. As we see in Figure 2.8, these constraints are specified by the implementation code for the network database.

We have also used a separate record type (EMP_NAME) to implement the subrecord $Name$ which is part of the $Employee$ record. The owner-member set EMP_NAMES which we use to relate employees and their names, is implemented as one-to-one relationship type in Figure 2.8.

2.2.7 Maintaining the Network Database

Network database applications were traditionally implemented in Cobol or Pascal, such that a separate record was used for each record and set type of the network database schema. E.g., in a Pascal application interface a record variable for $Employee$s could be defined as follows:

```
SCHEMA NAME IS COMPANY

RECORD NAME IS DEPARTMENT
  DUPLICATES ARE NOT ALLOWED FOR DNUMBER
  DUPLICATES ARE NOT ALLOWED FOR NAME
    DNUMBER         TYPE IS   NUMERIC INTEGER
    NAME            TYPE IS   CHARACTER 20
    LOCATIONS       TYPE IS   CHARACTER 20 VECTOR

RECORD NAME IS MANAGES
  DUPLICATES ARE NOT ALLOWED FOR DEPTNUMBER
  DUPLICATES ARE NOT ALLOWED FOR MGRSSN
    DEPTNUMBER      TYPE IS   NUMERIC INTEGER
    MGRSSN          TYPE IS   NUMBERIC 9
    MGRSTARTDATE    TYPE IS   CHARACTER 11

RECORD NAME IS SUPERVISINGDEPARTMENT
  DUPLICATES ARE NOT ALLOWED FOR SD_NUMBER
    SD_NUMBER          TYPE IS   NUMERIC INTEGER

RECORD NAME IS EMPLOYEE
  DUPLICATES ARE NOT ALLOWED FOR SSN
    SSN             TYPE IS   NUMBERIC 9
    BIRTHDATE       TYPS IS   CHARACTER 11
    ADDRESS         TYPE IS   CHARACTER 30
    SEX             TYPE IS   CHARACTER 1
    DEPTNAME        TYPE IS   CHARACTER 20
    SALARY          TYPE IS   NUMERIC (8,1)

RECORD NAME IS EMP_NAME
  DUPLICATES ARE NOT ALLOWED FOR FIRSTNAMES, LASTNAME
    FIRSTNAMES      TYPE IS   CHARACTER 15  VECTOR
    LASTNAME        TYPE IS   CHARACTER 20
    TITLE           TYPE IS   CHARACTER 10

RECORD NAME IS PROJECT
  DUPLICATES ARE NOT ALLOWED FOR PNUMBER
    PNUMBER         TYPE IS   NUMBERIC (10,2)
    NAME            TYPS IS   CHARACTER 30
    LOCATION        TYPE IS   CHARACTER 20
    PROJSTARTDATE   TYPE IS   CHARACTER 11
    DURATION        TYPE IS   NUMERIC (5,1)

RECORD NAME IS HAS_PARTICIPANTS
  DUPLICATES ARE NOT ALLOWED FOR EMPSSN, PNUMBER
    EMPSSN          TYPE IS   NUMBERIC 9
    PNUMBER         TYPE IS   NUMBERIC (10,2)
    HOURS           TYPE IS   NUMERIC (5,1)

SET NAME IS ALL_DEPTS
  OWNER IS SYSTEM
    ORDER IS SORTED BY DEFINED KEYS
  MEMBER IS DEPARTMENT
    KEY IS ASCENDING DNUMBER

SET NAME IS SUPERVISES
  OWNER IS SUPERVISINGDEPARTMENT
  MEMBER IS DEPARTMENT
    KEY IS ASCENDING DNUMBER
    INSERTION IS MANUAL

SET NAME IS EMP_NAMES
  OWNER IS EMPLOYEE
    ORDER IS SYSTEM DEFAULT
    DUPLICATES ARE NOT ALLOWED
  MEMBER IS EMP_NAME
    KEY IS FIRSTNAMES, LASTNAME
    INSERTION IS AUTOMATIC
    RETENTION IS MANDATORY

SET NAME IS EMPLOYS
  OWNER IS DEPARTMENT
    ORDER IS SORTED BY DEFIBED KEYS
  MEMBER IS EMPLOYEE
    INSERTION IS AUTOMATIC
    RETENTION IS MANDATORY
    CHECK IS DEPTNAME IN EMPLOYEE =
                          NAME IN DEPARTMENT

SET NAME IS E_MANAGES
  OWNER IS EMPLOYEE
    DUPLICATES ARE NOT ALLOWED
  MEMBER IS MANAGES
    INSERTION IS MANUAL
    RETENTION IS OPTIONAL
    DUPLICATES ARE NOT ALLOWED
    SET SELECTION IS BY APPLICATION

SET NAME IS D_MANAGES
  OWNER IS DEPARTMENT
    KEY IS NAME
  MEMBER IS MANAGES
    INSERTION IS MANUAL
    RETENTION IS OPTIONAL
    DUPLICATES ARE NOT ALLOWED

SET NAME IS E_WORKS_ON
  OWNER IS EMPLOYEE
  MEMBER IS HAS_PARTICIPANTS
    INSERTION IS AUTOMATIC
    RETENTION IS FIXED
    DUPLICATES ARE NOT ALLOWED
    SET SELECTION IS BY APPLICATION

SET NAME IS P_HAS_PARTICIPANTS
  OWNER IS PROJECT
  MEMBER IS HAS_PARTICIPANTS
    INSERTION IS MANUAL
    RETENTION IS OPTIONAL
    DUPLICATES ARE NOT ALLOWED

SET NAME IS HAS_LEADER
  OWNER IS PROJECT
    DUPLICATES ARE NOT ALLOWED
  MEMBER IS EMPLOYEE
    INSERTION IS MANUAL
    RETENTION IS OPTIONAL

SET NAME IS CONTROLS
  OWNER IS DEPARTMENT
  MEMBER IS PROJECT
    INSERTION IS MANUAL
    RETENTION IS OPTIONAL
    DUPLICATES ARE NOT ALLOWED
```

Figure 2.8: Definition of the Record- and Set-Types for the Company Database.

```
type
 FirstnameRecord =
  record
    FIRSTNAME: packed array [1..15] of char;
    next: ^FirstnameRecord
  end;
 EmpNameRecord =
  record
    FIRSTNAMES: ^FirstnameRecord;
    LASTNAME: packed array [1..20] of char;
    TITLE: packed array [0..10] of char
  end;
var
 employee:
  record
    SSN: packed array [1..9] of char;
    NAME: EmpNameRecord;
    BIRTHDATE: packed array [1..11] of char;
    ADDRESS: packed array [0..30] of char;
    SEX: char;
    SALARY: real
  end
```

Then, the following Pascal fragment with embedded statements of the network data manipulation language (DML) can be used to fetch the employee "Jon Smith" and print his birthdate:

```
with employee.NAME
do begin
  new(FIRSTNAMES); FIRSTNAMES->FIRSTNAME := 'John'; FIRSTNAMES->next := nil;
  LASTNAME := 'Smith'
end;
$FIND ANY employee.NAME WITHIN EMP_NAMES USING FIRSTNAMES, LASTNAME;
if DB_STATUS = 0
then begin
  (* the EMP_NAMES cursor as well as the EMPLOYEE cursor *)
  (* are now on the emp_name and employee record "Jon Smith" *)
  $FIND ANY EMPLOYEE;
  if DB_STATUS = 0
  then begin
    $GET employee;
    writeln(employee.NAME.FIRSTNAMES->FIRSTNAME,' ',
            employee.NAME.LASTNAME,' was born:');
    writeln(employee.BIRTHDATE)
  end
end
```

So far, to give the reader an impression of realization and application implementation on network databases. But, for reason of space further details like the *STORE* and *MODIFY* commands are ommited here. If he wishes to know more about these things, the interested reader is directed to according publications, Codasyl [DTG71, DTG79] or, for example, [Oll78]. Chapter 10 in [EN94] also gives examples to understand implementation techniques for network databases. Network DBMSs are, for instance, IBM's *IDMS* or Siemens's *UDS*. The code presented in this Section is valid for the IDMS system.

Properties which are important for this work. The essential aspect for this work is to see the navigational semantics of hierarchical and network database applications, processing each record as a separate item, which is reasoned by the underlying data model and the database management system implemented on top of this model. This makes them inefficient for evaluations on record sets, like, for example, determining the average age of the employees or summing up their salaries.

2.3 The Relational Data Model (Relational Model)

In contrast to the hierarchical data model and the network model, the relational data model specifies properties of data based on record sets, and not on the special records. Since it fundamentally bases on the following three concepts only, the relational model [Cod70, Cod79] is sometimes looked at to be (too) simple in structure:

1. attributes, which are of atomar domains, such as boolean, integer, character, but may not be composite or multivalued, that is, they must not be records, lists, or sets;

2. relation schemata (~record types), which represent objects in the real world or relations between them, and are sequences of attributes;

3. and relations, which are sets of records (or tuples) that are described by the relation schemata.

Besides these structural concepts, the relational data model *(relational algebra)* defines a set of operators:

1. Projection (π). The projection of a relation (table) R on the attribute sequence X is defined by considering only the attributes (columns) of X in that order. In relational algebra, the projection is described by a term like $\pi_X(R)$. However, we will use $R[X]$ to describe the projection of relation R on the attributes X.

2. Selection (σ). The selection of a relation R according a predicate p generates the set of all tuples of R for which p holds. We write $\sigma_p(R)$.

3. Join ($\bowtie$). The join between two relations, R_1 and R_2, with an equal attribute set X in their schemata, and S_1 is the relation schema of R_1, S_2 is the relation schema of R_2, means to build the product of all tuples $t_1 \in R_1$ and $t_2 \in R_2$ such that $t_1[X] = t_2[X]$, and finally projecting the result set on $S_1 \cup (S_2 \backslash X)$. We write $R_1 \bowtie R_2$.

Furthermore, the relational data model presupposes that there are *generic database retrieval and update operations*, like select, insert, delete, and update, in SQL. Generic

means that they are applicable for all relations in the database and are used to change their contents, such that no special operations for the different sets need to be designed. (Note that we use "operator" and "operation" in a different meaning here: an operator is meant to deliver a term which can be used in further terms, whereas an operation is understood as a procedure that invokes some status change on the objects with which it works.)

2.3.1 Integrity Constraints

Integrity constraints have special meaning according relational databases because they are essential for the design of relation schemata. In this section we consider *static integrity constraints*; we call integrity constraints *static* if they are considering only single transactions– like the insert, delete and update operation (in SQL) –which lead either to a correct or else to an incorrect database state, and so, are commited or aborted (rolled back). On the other hand, we call integrity constraints *dynamic* if they are used for the representation of the behavior of the database during its lifetime. This way, composite transactions, such as an insert operation that triggers another insert operation, are considered by *dynamic integrity constraints* which will be described in Section 3.3.2.1.

Static integrity constraints are classified into *domain constraints* and *data dependencies*.

2.3.1.1 Domain Constraints

Domain constraints restrict the domain of attributes, such as the age of a person, which must not be negative. They are defined on min/max values or on a special discrete set, such as "month" which must be in {Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec}.[2]

Convention. *In the following, we use a,b,c,... as denoters for attributes, X,Y,Z for attribute sets (or for attribut sequences), t (with underscripts) for tuples, R (with underscripts) for relations, and S (with underscripts) for relation schemata, respectively.*

2.3.1.2 Functional Dependencies (FDs)

A functional dependency $X \rightarrow Y$ defines a mapping between the instances of two attribute sets, such that each distinct value (tuple) of the first set (X) uniquely determines a value (tuple) of the second set (Y).

Definition (Functional Dependency). Let R be a relation, S be the schema of R, and X,Y $\subseteq S$, then

[2] Today, domain constraints in SQL are usually implemented by $CHECK$ clauses which are appended to the column definitions of the create table statements.

$$X \rightarrow Y ::=$$
$$\forall\, t_1, t_2 \in R: \qquad t_1[X]{=}t_2[X] \Longrightarrow t_1[Y]{=}t_2[Y].$$

That is, whenever two tuples t_1 and t_2 of a relation R with attributes (relation schema) S, such that X,$Y \subseteq S$ match on X, then they also match on Y. Recall the ER-Schema in Figure 2.1. Table 2.1 shows a relation which represents the occurence set of the *Employee* entity type.

Ssn	Firstname	Lastname	Title	Birthdate	Address	Sex	Salary
12349875	David	Miller	Dr.	1966/04/23	Berlin	m	117,000
78654312	Sven	Martin		1965/04/15	Paris	m	98,000
23456798	Mary-Ann	Miller		1965/04/15	Berlin	f	98,000
34215672	Jon	Smith		1949/11/17	Kingston	m	250,000

Table 2.1: Employee Relation.

Here, the *Employee*'s social security number (*Ssn*) determines his *Firstname*, *Lastname*, *Title*, *Birthdate*, *Address*, *Sex* and *Salary*, the *Employee*'s *Firstname* and *Lastname* together determine his *Ssn*, *Firstname*, *Lastname*, *Title*, *Address*, *Sex* and *Salary*, and the *Employee*'s *Lastname* determines his *Address*, such that we have:

- $\{Ssn\} \rightarrow \{Firstname,Lastname,Title,Birthdate,Address,Sex,Salary\}$
- $\{Firstname,Lastname\} \rightarrow \{Ssn,Title,Birthdate,Address,Sex,Salary\}$
- $\{Lastname\} \rightarrow \{Address\}$

This way, another representation of the database could be given by the relations shown in Table 2.2.

Ssn	Firstname	Lastname	Address
12349875	David	Miller	Berlin
78654312	Sven	Martin	Paris
23456798	Mary-Ann	Miller	Berlin
34215672	Jon	Smith	Kingston

Firstname	Lastname	Title	Birthdate	Sex	Salary
Jon	Smith		1949/11/17	m	250,000
David	Miller	Dr.	1966/04/23	m	117,000
Mary-Ann	Miller		1965/04/15	f	98,000
Sven	Martin		1965/04/15	m	98,000

Table 2.2: Decomposing the Employee Relation.

However, the third functional dependency, $\{Lastname\} \rightarrow \{Address\}$, is not given when a further tuple is added to the relation, as shown in Table 2.3.

Ssn	Firstname	Lastname	Title	Birthdate	Address	Sex	Salary
12349875	David	Miller	Dr.	1966/04/23	Berlin	m	117,000
78654312	Sven	Martin		1965/04/15	Paris	m	98,000
23456798	Mary-Ann	Miller		1965/04/15	Berlin	f	98,000
34215672	Jon	Smith		1949/11/17	Kingston	m	250,000
78124367	Susan	Smith		1957/07/12	New York	f	140,000

Table 2.3: Adding Another Record to the Employee Relation.

Axioms for Functional Dependencies and Decompositions of Relation Schemata. Functional dependencies are the fundamental of *lossless decompositions* of relation schemata. The axioms *(Amstrong axioms)* which hold for functional dependencies are informally described by Table 2.4.

Decomposition and Composition. Many authors consider functional dependencies only in the from $\{X\} \rightarrow a$ such that the right-hand-side is a single attribute. But, we consider here the right-hand-sides as sets, such that functional dependencies can be decomposed and composed without loss of information. That is

1. from $X \rightarrow \{a,b,c,d,...\}$ we can derive $X \rightarrow \{a\}$, $X \rightarrow \{b\}$, and so forth,
2. and, if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow Y \cup Z$ holds as well.

Inclusion. For each attribute set X and attribute set Y such that $Y \subseteq X$
$X \rightarrow Y$ holds.

Augmentation. For any attribute a:
if $X \rightarrow Y$, then $X \cup \{a\} \rightarrow Y$.

Transitivity. Transitivity holds for functional dependencies:
$X \rightarrow Y \wedge Y \rightarrow Z \Longrightarrow X \rightarrow Z$.

Table 2.4: Axioms for Functional Dependencies.

From the axioms, we can evaluate minimal covers of the sets of functional dependencies (FDs). E.g. there exists an algorithm that evaluates a minimal set of functional dependencies from a given FD set, such that all functional dependencies (the closure) can be evaluated from the minimal set, by using the axioms in Figure 2.4. E.g., for the FDs of the relation in Figure 2.3, $\{Ssn\} \rightarrow \{Firstname, Lastname, Title, Birthdate, Address, Sex, Salary\}$ and $\{Firstname, Lastname\} \rightarrow \{Ssn, Title, Birthdate, Address, Sex, Salary\}$, we obtain the minimal set of functional dependencies

- $\{Ssn\} \rightarrow \{Firstname, Lastname\}$, and
- $\{Firstname, Lastname\} \rightarrow \{Title, Birthdate, Sex, Salary\}$.

The decomposition of the *Employee* relation according these FDs is shown in Figure 2.5.

R_1

Ssn	Firstname	Lastname
12349875	David	Miller
78654312	Sven	Martin
23456798	Mary-Ann	Miller
34215672	Jon	Smith
78124367	Susan	Smith

R_2

Firstname	Lastname	Title	Birthdate	Address	Sex	Salary
Jon	Smith		1949/11/17	Kingston	m	250,000
Susan	Smith		1957/07/12	New York	f	140,000
David	Miller	Dr.	1966/04/23	Berlin	m	117,000
Mary-Ann	Miller		1965/04/15	Berlin	f	98,000
Sven	Martin		1965/04/15	Paris	m	98,000

Table 2.5: Decomposing the Employee Relation according Functional Dependencies.

Since now *joining* the two relations of the decomposition, such that t_1 is a tuple of relation R_1, t_2 is a tuple of relation R_2, and

$$t_1[Firstname,Lastname] = t_2[Firstname,Lastname] \ ,$$

restores the old relation (Table 2.3), this decomposition is also denoted *lossless*.

Key Dependencies (KDs). A *key dependency* is a special functional dependency, such that the left-hand-side determines all attributes of the relation schema. Considering the first example relation of this Section (Table 2.1) with its functional dependencies,

1. $\{Ssn\} \rightarrow \{Firstname,Lastname,Title,Birthdate,Address,Sex,Salary\}$,
2. $\{Firstname,Lastname\} \rightarrow \{Ssn,Title,Birthdate,Address,Sex,Salary\}$,
3. and $\{Lastname\} \rightarrow \{Address\}$,

the 1st FD and the 2nd FD are key dependencies, whereas the 3th FD ($\{Lastname\} \rightarrow \{Address\}$) is not a key dependency. Considering the second example relation (Table 2.3) both functional dependencies are key dependencies.

In the decomposition of the second relation (Table 2.5) we did choose $\{Ssn\}$ as the key of R_1

- thus creating the functional key dependency
  $\{Ssn\} \rightarrow \{Firstname,Lastname\}$,

and $\{Firstname,Lastname\}$ as the key of R_2

- thus creating the functional key dependency
  $\{Firstname,Lastname\} \rightarrow \{Title,Birthdate,Address,Sex,Salary\}$.

However, as one might infer from the axioms in Table 2.4, this is not the only possibility for the lossless decomposition of the relation in Table 2.5.

2.3.1.3 Inclusion Dependencies (IDs)

Functional dependencies (FDs) are considered as *intrarelational*. Inclusion dependencies, on the other hand, are considered as *interrelational*; that is, they specify equalities between values (or combinations of values) which are contained in different relations. An inclusion dependency $R_x[X] \subseteq R_y[Y]$ specifies that all tuples in $R_x[X]$ must be contained in $R_y[Y]$.

Definition (Inclusion Dependency). Let R_x, R_y be relations, S_x be the schema of R_x, S_y be the schema of R_y, such that $X \subseteq S_x$ and $Y \subseteq S_y$, and the attributes of the sequences X and Y are pairwise type-compatible, then

$$R_x[X] \subseteq R_y[Y] ::= \\ \forall\, t_x \in R_x: \quad \exists\, t_y \in R_y:\, t_y[Y] = t_x[X]$$

Consider relation R_1 and relation R_2 in Table 2.5. Here, the following inclusion dependencies are given

- $R_1[Firstname,Lastname] \subseteq R_2[Firstname,Lastname]$, and
- $R_2[Firstname,Lastname] \subseteq R_1[Firstname,Lastname]$,

because the tuple sets of R_1 and R_2 are equal on the attributes $[Firstname,Lastname]$.

Such mutual inclusion dependencies are rather unfrequent. Therefore, let us consider the following example: A *Manager is an Employee*, and the *Manager* relation has the schema $\{Ssn,DeptNumber\}$ and key $\{Ssn\}$. Thus, we obtain $Manager[Ssn] \subseteq Employee[Ssn]$, but $Employee[Ssn] \subseteq Manager[Ssn]$ does not hold.

Referential Dependencies (REFs). Since relational databases are value-based, referential dependencies are value-based as well– unlike the pointers of hierarchical databases and network databases. Therefore, *referential dependency* and *inclusion dependency* in relational databases are often used as synonyms, because the values of $t_x[X]$ (t_x is tuple in R_x) are used as reference to a tuple t_y in R_y ($t_x[X] = t_y[Y]$). The latter mentioned ID $Manager[Ssn] \subseteq Employee[Ssn]$ is a typical referential dependency.

2.3.2 Normal Forms for Relational Schemata

Normal forms are describing the transformation of a single relation schema into a set of relation schemata such that

- the mapping of data is non-redundant (the same relations between data are not repeated),
- functional dependencies are controlled by the structuring of the new schemata,
- and operations (insert, delete, update) do not generate undesired results, according that FDs are violated.

2.3.2.1 First Normalform (1NF)

One of the most fundamental assumptions of the relational data model is that data is represented in the form of flat tables, called *first-normalform assumption.* This way a relation attribute is not allowed to carry values of records, lists, or sets, for instance. However relational database implementations which use repeating groups of attributes, e.g. $Firstname1$, $Firstname2$, and $Firstname3$, are sometimes considered to be not 1NF, although this is not really true– because of the atomar domains (of $Firstname1$, $Firstname2$, and $Firstname3$, respectively).

$(NF)^2$ Relations. Relation schemata which really use record-typed, list-typed, or set-typed attributes, are called *non-first-normal-form* ($(NF)^2$). As we will see in Section 3.3.2, normalization can also take place for such $(NF)^2$ relations. However, we will characterize the traditional normal forms firstly.

2.3.2.2 Second Normalform (2NF)

Second normalform (2NF) is based on the concept of full functional dependency. Consider the left relation in Table 2.2, which contains the functional dependencies $\{Ssn\} \rightarrow \{Firstname,Lastname\}$ and $\{Lastname\} \rightarrow \{Address\}$. Since $\{Lastname\}$ is not a key for that relation, the functional key dependency $\{Ssn,Lastname\} \rightarrow \{Address\}$ is contained in the relation as well, such that $\{Address\}$ is not anymore fully functional dependent on the candidate key $\{Ssn,Lastname\}$. Therefore, this relation is said to be not 2NF. Second normalform (2NF) requires relations to contain no partial FDs and to use minimal keys.

2.3.2.3 Third Normalform (3NF)

Third normalform (3NF) requires that no transitive functional dependencies are contained in a single relation schema. Consider again the relation shown in Table 2.3. From the minimal set of functional dependencies for this relation, $\{Ssn\} \rightarrow \{Firstname,Lastname\}$ and $\{Firstname,Lastname\} \rightarrow \{Title,Birthdate,Address,Sex,Salary\}$ we can derive $\{Ssn\} \rightarrow \{Title\}$, which is transitive and included in that relation. Therefore, the relation in Table 2.3 is not 3NF. However, the relations R_1 and R_2 in Table 2.5 are 3NF, because no transitive functional dependency is contained in one of the relations, R_1 or R_2. 3NF can be derived for each relation schema R and set of functional dependencies F using the algorithm shown in Table 2.6.

2.3.2.4 Boyce-Codd Normalform (BCNF)

Boyce-Codd normalform (BCNF) restricts each functional dependency to be a key dependency *(the left-hand-side of each functional dependency must be a superkey for the relation*

1. Find a minimal cover F_{min} from the set of functional dependencies F.
2. As long as there are $f_1 = A \rightarrow B$ and $f_2 = A \rightarrow C$ in F_{min} (with equal left-hand-sides) add $A \rightarrow B \cup C$ to F_{min}, and delete f_1 and f_2.
3. Create a separate relation for each functional dependency in F_{min}; that is, foreach $A \rightarrow B \in F_{min}$ create a relation based on the schema, $A \cup B$, and the key A. Let the set of the new relations be RS.
4. If there is no relation schema R' in RS, such that the key of R' functionally determines (directly or indirectly, by the axioms in Table 2.4) all attributes in R, then add a relation to RS such that its schema is a common key for all attributes of the relations in RS. That is, unite all keys (sets of key attributes) of the relation schemata in RS, whose key is not contained in any set of non-key attributes (right-hand-side of a functional dependency in F_{min}) of some other relation schema in RS. The union of those keys is used to create a separate relation, whose key is the whole schema of that relation.
5. Delete all relations R_1 from RS, whose schema is contained in another R_2 in RS, and attach the functional dependency from which R_1 was constructed to R_2.

Table 2.6: Algorithm which derives a minimal set of Relation Schemata that is 3NF.

schema in which the attributes of the functional dependency are included). BCNF includes 3NF, but, presupposing that the set of normalized result schemata does not contain any relation schema that is included in another relation schema in that set, BCNF can not be derived for each relation schema and set of functional dependencies (as it is possible for 3NF). Assume a relation schema R is given by the attribute set $\{a,b,c\}$, and the set of functional dependencies is $\{\{a,b\} \rightarrow \{c\}, \{c\} \rightarrow \{a\}\}$. Thus, R is in 3NF, but not in BCNF, because $\{c\}$ of the FD $\{c\} \rightarrow \{a\}$ is not a superkey of R.
For this example, the 3NF normalization algorithm in Table 2.6 generates firstly two relation schemata, $R_1 = \{a,b,c\}$ and $R_2 = \{c,a\}$, but then deletes R_2 (by the 5th step), since its attribute set is included in R_1, such that we have the non-(super)key functional dependency $\{c\} \rightarrow \{a\}$ in R_1 again.

2.3.3 Further Normalization

2.3.3.1 Multivalued Dependencies (MDs)

For reason of the *first-normalform assumption,* the relational model does not support domains for more than one value, such as list and set. To consider such non-atomic

values as well, Fagin introduced multivalued dependencies (MDs) in 1977 [Fag77]. MDs are specifying mappings between attributes X and Y such that each tuple on X is related to a set of tuples on Y. Formally, an MD can be described as follows:

Definition (Multivalued Dependency). Let R be a relation, S be the schema of R, X,$Y \subseteq S$, $Z = S \backslash (X \cup Y)$ and $Z \neq \emptyset$, then

$$X \twoheadrightarrow Y ::=$$
$$\forall\, t_1,t_2 \in R: \quad t_1[X]=t_2[X] \wedge t_1[Y] \neq t_2[Y] \wedge t_1[Z]=t_2[Z]$$
$$\wedge\, \exists\, t_3 \in R: \quad t_3[X]=t_1[X] \wedge t_3[Y]=t_1[Y] \wedge t_3[Z] \neq t_1[Z] \implies$$
$$\exists\, t_4 \in R: \quad t_4[X]=t_1[X] \wedge t_4[Y]=t_2[Y] \wedge t_4[Z]=t_3[Z]$$

A functional dependency can be considered as a special case of multivalued dependency, such that the instance of the attribute set of the right-hand-side is only a singleton set.

Equality-generating and Tuple-generating Dependencies. FDs are considered as *equality-generating* dependencies, since they specify restrictions on the equality of tuples that must be given by a concrete database instance at any time. IDs and MDs require the existence of additional tuples in database relations. They are therefore called *tuple-generating*.

2.3.3.2 Fourth Normalform (4NF)

Fourth normalform (4NF) was developed to resolve these problems of additional tuples that have to exist or must be generated:

> A relation R with schema S is in 4NF, if for each non-trivial MD $X \twoheadrightarrow Y$ in F^+, which is the cover of the functional and multivalued dependencies F, of the dependencies defined for R, X is a superkey of R.

This way, relations of a 4NF database must have only attributes of a functional or multivalued dependency; there must not be a relation that has more than one multivalued dependency, such that the tuple-generating property of an MD is implicitely given. For reason that an FD is a special case of an MD restricting that the right-hand-side attribute set has only instances which are singleton sets, a relation schema that is 4NF is also BCNF.

2.3.3.3 Join Dependencies (JDs)

A join dependency (JD) is based on the fact, whether the join of sub-relations ($R_1 \bowtie R_2 \bowtie ...$) which are won by decomposing a relation R into R_1,R_2,... restores R or not. If R is decomposed into R_1,R_2,R_3,R_4 and $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 = R$ then the join dependency $JD(R,\{R_1,R_2,R_3,R_4\})$ is said to hold.

Multivalued dependencies (MDs) are a special case of join dependency, such that a relation is decomposed into two sub-relations only. So, if there is a relation R with schema S and X,Y,$Z \subset S$, $Z = S \backslash (X \cup Y)$, $X \twoheadrightarrow Y$, then after decomposing R into R_1 with schema $X \cup Y$ and R_2 with schema $X \cup Z$, the join dependency $JD(R,\{R_1,R_2\})$ holds.

2.3.3.4 Fifth Normalform (5NF)

Fifth normalform (5NF) is given for a relation R if not any join dependency of sub-relations of R holds. Otherwise, to transform R into 5NF, it must be decomposed into the sub-relations for which the join dependency holds.

2.3.3.5 Domain-Key Normalform (DKNF)

The idea of domain-key normalform (DKNF) is to consider all possible constraints, which hold for a relation. That are domain constraints and data dependencies (FD,ID,MD,JD) as well. Besides the domain constraints considered in Section 2.3.1.1, domain constraints are also given by the equality-generating property of FDs, for instance. Consider the relation in Table 2.7, where δ_1 and δ_2 represent unknowns.

Firstname	Lastname	Department	Manager
David	Miller	Computer Science	Miller
Sven	Martin	Computer Science	Miller
Mary-Ann	Miller	Mathematics	Newman
Jon	Smith	Computer Science	δ_1
Susan	Smith	Mathematics	δ_2

Table 2.7: A Relation with unkowns.

From the relation we could assume the FD $\{Department\} \rightarrow \{Manager\}$, which could be derived from the upper two tuples. This way, we could assign the following values to the unknowns: δ_1="Miller" and δ_2="Newman".
In Table 2.8 we see another relation. Let us assume that for the relation in Table 2.8 the following MDs hold

$$\{Firstname,Lastname\} \twoheadrightarrow \{Project\} \text{ and } \{Firstname,Lastname\} \twoheadrightarrow \{Skill\}.$$

But then, presupping that the relation contains the seven upper tuples, the last two tuples $(Jon,Smith,Analysis,Engineering)$ and $(Jon,Smith,Analysis,Physics)$ must be contained as well.

On the other hand, if it is really known that the FD $\{Department\} \rightarrow \{Manager\}$ is valid for the relation in Table 2.7 and the two above MDs are valid for the relation in Table 2.8, then the equality-generation and tuple-generation must take place. The best way to do that is to normalize the relations according 3NF (Table 2.7) and 4NF (Table

Firstname	Lastname	Project	Skill
Sven	Martin	Development	Computers
Sven	Martin	Analysis	Computers
Sven	Martin	Development	Philosophy
Sven	Martin	Analysis	Philosophy
David	Miller	Documentation	Computers
David	Miller	Analysis	Computers
Mary-Ann	Miller	Analysis	Mathematics
Jon	Smith	Implementation	Engineering
Jon	Smith	Analysis	Physics
Susan	Smith	Analysis	Philosophy
Jon	Smith	Analysis	Engineering
Jon	Smith	Implementation	Physics

Table 2.8: A Relaion with possible MDs, which generate additional Tuples.

2.8): the relations are decomposed, such that the relations of the decomposition are in a normalized form, respectively. Table 2.9 and 2.10 show how the normalized relations look like.

Firstname	Lastname	Department
David	Miller	Computer Science
Sven	Martin	Computer Science
Mary-Ann	Miller	Mathematics
Jon	Smith	Computer Science
Susan	Smith	Mathematics

Department	Manager
Computer Science	Miller
Mathematics	Newman

Table 2.9: Normalizing the Relation of Table 2.7.

However, as we have seen by the example relations, normalization is not a panacea for database design; even though it deletes multiple functional and multivalued dependencies from a single relation schema it generates inclusion dependencies which must be additionally maintained be the DBMS.

2.3.4 Further Data Dependencies for Relational Databases

In Section 2.3.1 and 2.3.3 we have already considered functional dependencies (FDs), key dependencies (KDs), inclusion dependencies (IDs), referential dependencies (REFs), multivalued dependencies (MDs), and join dependencies (JDs). These dependencies are equality-generating or tuple-generating. We will now present three additional dependency types, such that the first one generally can not be classified as equality-generating or tuple-generating, and the last two ones are neither equality-generating nor tuple-generating.

Firstname	Lastname	Project
Sven	Martin	Development
David	Miller	Documentation
David	Miller	Analysis
Mary-Ann	Miller	Analysis
Jon	Smith	Implementation
Jon	Smith	Analysis
Susan	Smith	Analysis

Firstname	Lastname	Skill
Sven	Martin	Computers
Sven	Martin	Philosophy
David	Miller	Computers
Mary-Ann	Miller	Mathematics
Jon	Smith	Engineering
Jon	Smith	Physics
Susan	Smith	Philosophy

Table 2.10: Normalizing the Relation of Table 2.8.

The first type of data dependency which we will consider now, are *cardinality constraints*. The others are *exclusion dependencies* and *afunctional dependencies*, which may be characterized as *unequality-generating*.

2.3.4.1 Cardinality Constraints (CCs)

In this work we use *cardinality constraints* instead of *multiplicities,* which we had used in Figure 2.1. These may also be specified for entity-relationship schemata, and are here considered in the meaning of participation constraints, such that $card(R_1,R_2) = (m,n)$ with relations R_1 and R_2 expresses that each member of relation R_2 has at least m and at most n associated tuples in relation R_1. Whenever we use cardinality constraints in the graphical schema (entity-relationship schema, relational schema, or as we will see, object model) we draw the cardinality (m,n) on the line or arrow between R_1 and R_2, such that it is closer to the second relation type (respectively entity type, relationship type, or class), R_2.

2.3.4.2 Exclusion Dependencies (EDs)

An exclusion dependency (ED) describes that the value sets (or tuple sets) of two relations– or projections of relations –are mutual exclusive. Assume the mutual exclusion is for the attribute sequence X of relation R_x and the attribute sequence Y of relation R_y, which is denoted by the term $R_x[X] \| R_y[Y]$.

Definition (Exclusion Dependency). Let R_x, R_y be relations, S_x be the schema of R_x, S_y be the schema of R_y, $X \subseteq S_x$, $Y \subseteq S_y$, and the attributes of the sequences X and Y are pairwise type-compatible, then

$$R_x[X] \parallel R_y[Y] ::= \\ \forall\, t_x \in R_x\colon \quad \nexists\, t_y \in R_y\colon \quad t_y[Y] = t_x[X]$$

That tuples which are in $R_y[Y]$ must not be in $R_x[X]$ ($\forall\, t_y \in R_y\colon \nexists\, t_x \in R_x\colon t_x[X] = t_y[Y]$) is given implicitely. This way the tuple sets are distinct ($R_x[X] \cap R_y[Y] = \emptyset$).

2.3.4.3 Afunctional Dependencies (ADs)

Functional dependencies ($X \rightarrow Y$) describe a unique mapping from the attribute set of the left-hand-side (X) to the attribute set of the right-hand-side (Y). This restricts tuples to be equal on Y whenever tuples are equal on X. An afunctional dependency (AD) describes the opposite case. That is, two tuples t_1,t_2 in a relation R must not be equal on Y whenever they are equal on X. Formally, an afunctional dependency ($X \not\rightarrow Y$) is specified by:

Let R be a relation, S its schema, X,$Y \subset S$.
$X \not\rightarrow Y$ ::=
$$\forall\, t_1,t_2 \in R,\ t_1 <> t_2:\quad t_1[X]=t_2[X] \Longrightarrow t_1[Y] \neq t_2[Y].$$

However, since ADs are specially considered by the *semantics acquisition* part of the RADD project (see for instance [Alb94])), they are not looked at any further here.

The unequality-generating property of EDs and ADs makes it not possible to implement them by the structure of the database such that they were considered by normalization and a special normal form.

2.3.5 Relational Database Implementation

Recall the ER representation of the Company Schema (Figure 2.1) and the network implementation schema of it (Figure 2.7). Figure 2.9 shows the relational database schema that implements the Company database.

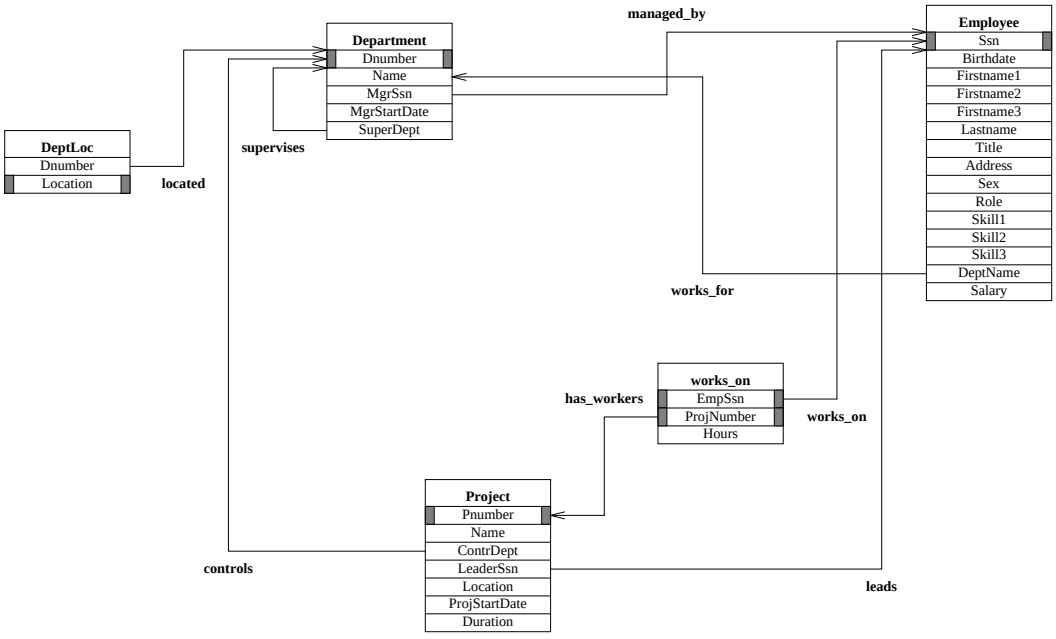


Figure 2.9: The Relational Schema for Implementation of the Company Database.

2.3.5.1 Properties of the Relational Database Implementation

As mentioned above, the schema of a relational database need not necessarily to be hierarchical. So, Figure 2.9 contains the relation *Employee* which has a foreign-key to *Department* (*DeptName*) which represents his *works_for* property, and reversally, *Department* has with the *Department Manager* attribute (*MgrSsn*) a foreign-key to *Employee*. *Department* has another foreign-key attribute (*SuperDept*) which is a reference to the *Department* which supervises that *Department* (if there is one). This way, the relational code (SQL-code) for the definition of these tables, their keys and foreign-keys, as well as for definition of the indices, views, and triggers can be given as shown in Figure 2.10.

In the *CREATE TABLE* statements for the *Department* relation in Figure 2.10, we have left the *MgrSsn* attribute *with null* (there is no NOT NULL clause for that attribute), such that it is possible to insert the first *Department* without having already the *Employee* who is the *Department*'s manager:

```
insert into Department (Dnumber,Name)
values ('11.3.1','Computer Science');
```

Subsequently it is possible to insert the *Department*s which are dependent on the Computer Science department, e.g.:

```
insert into Department (Dnumber,Name,SuperDept)
values ('11.3.2','Software Development','11.3.1');
```

We have also included the attribute *Role* in the *Employee* relation which is used to represent the according role of the employee. The allowed values for the *Role* attribute are 'm', 'l', 'w', 's', and 'a', which is ensured by the CHECK clause in the lines which specify that attribute. For these roles we have created the *updatable* VIEWs *DeptManager*, *Secretary*, and *Assistant*. Views are updatable if they are specified for only one relation (such that they do not formulate a join or contain sub-selects). *ProjLeader* and *ProjWorker* are not updatable. Views are *insertable* if all mandatory attributes (NOT NULL attributes) are included in the select statement's attribute list. In this example, the WITH CHECK OPTION clause of the view definitions for *DeptManager*, *Secretary*, and *Assistant* automatically takes care that the correct value is used whenever an insert statement is issued according that view. Hence, it is possible to specify the command

```
insert into DeptManager
 (Ssn,Role,Birthdate,Sex,Firstname1,Lastname,Title,Skill1,DeptName,Salary)
values
 ('123456789','m','1996/04/23','m','David','Miller','Dr.',
  'Computers','Computer Science','117000');
```

which the DBMS handles as if the following command were used:

```
CREATE TABLE Department (
  Dnumber DECIMAL(6,2) NOT NULL,
  Name CHAR(20) NOT NULL,
  MgrSsn DECIMAL(9),
  MgrStartDate DATETIME,
  SuperDept DECIMAL(6,2),
  CONSTRAINT DeptPK_a1 PRIMARY KEY (Dnumber),
  CONSTRAINT DeptFK_a6 FOREIGN KEY (SuperDept)
                        REFERENCES Department(Dnumber)
);

CREATE UNIQUE INDEX DeptUX_a2 ON Department(Name);

REVOKE UPDATE ON Department.Dnumber,Department.Name
  FROM ALL;

CREATE TABLE DeptLoc (
  Dnumber DECIMAL(6,2) NOT NULL,
  Location VARCHAR(20) NOT NULL,
  CONSTRAINT DLocPK_a2 PRIMARY KEY (Location),
  CONSTRAINT DLocFK_a1 FOREIGN KEY (Dnumber)
                        REFERENCES Department(Dnumber)
   ON DELETE CASCADE
);

CREATE INDEX DLocIX_a1 ON DeptLoc(Dnumber);

CREATE TABLE Employee (
  Ssn DECIMAL(9) NOT NULL,
  Role CHAR(1) NOT NULL
   CHECK (Role IN ('m','l','w','s','a')),
  Birthdate DATETIME NOT NULL,
  Sex CHAR(1) CHECK (Sex IN ('m','f')),
  Firstname1 VARCHAR(15) NOT NULL,
  Firstname2 VARCHAR(15),
  Firstname3 VARCHAR(15),
  Lastname VARCHAR(20) NOT NULL,
  Title VARCHAR(10),
  Address VARCHAR(40),
  Skill1 VARCHAR(20) NOT NULL,
  Skill2 VARCHAR(20),
  Skill3 VARCHAR(20),
  DeptName VARCHAR(20) NOT NULL,
  Salary FLOAT NOT NULL,
  CONSTRAINT EmpPK_a1 PRIMARY KEY (Ssn)
);

CREATE INDEX EmpIX_a25 ON Employee(Firstname1,Lastname);
CREATE INDEX EmpIX_a15 ON Employee(DeptName);

CREATE TRIGGER EmpTriggerDept
  AFTER INSERT OR UPDATE ON Employee FOR EACH ROW
DECLARE dummy VARCHAR(20);
BEGIN
  SELECT Name INTO dummy FROM Department
   WHERE Name = :NEW.DeptName;
  IF SQLCODE <> 0 THEN
     raise_application_error(-11179, 'Department ' ||
                  :NEW.DeptName || ' does not exist!');
  END IF;
END;

CREATE TABLE Project (
  Pnumber DECIMAL(12) NOT NULL,
  Name VARCHAR(30) NOT NULL,
  ContrDept DECIMAL(6,2) NOT NULL,
  LeaderSsn DECIMAL(9) NOT NULL,
  Location VARCHAR(20) NOT NULL,
  ProjStartDate DATETIME,
  Duration SMALLFLOAT,
  CONSTRAINT ProjPK_a1 PRIMARY KEY (Pnumber),
  CONSTRAINT ProjFK_a3 FOREIGN KEY (ContrDept)
                        REFERENCES Department(Dnumber),
  CONSTRAINT ProjFK_a4 FOREIGN KEY (LeaderSsn)
                        REFERENCES Employee(Ssn)
);

CREATE INDEX ProjIX_a2 ON Project(Name);

CREATE TABLE works_on (
  EmpSsn DECIMAL(9) NOT NULL,
  ProjNumber DECIMAL(12) NOT NULL,
  Hours SMALLFLOAT,
  CONSTRAINT woPK_a12 PRIMARY KEY (EmpSsn,ProjNumber),
  CONSTRAINT woFK_a1 FOREIGN KEY (EmpSsn)
                      REFERENCES Employee(Ssn),
  CONSTRAINT woFK_a2 FOREIGN KEY (ProjNumber)
                      REFERENCES Project(Number)
);

ALTER TABLE Depparmemt ADD (
  CONSTRAINT DeptFK_a3 FOREIGN KEY MgrSsn
                        REFERENCES Employee(Ssn);
);

CREATE INDEX woIX_a2 ON works_on(ProjNumber);

CREATE TRIGGER handle_dept_ins
  AFTER INSERT ON Department
BEGIN
  IF :NEW.MgrSsn IS NOT NULL THEN
    UPDATE Employee SET Role = 'm' WHERE Ssn = :NEW.MgrSsn;
  END IF;
END;

CREATE TRIGGER handle_dept_del
  AFTER DELETE ON Department
  FOR EACH ROW
BEGIN
  IF :OLD.MgrSsn IS NOT NULL THEN
    UPDATE Employee SET Role = 'a' WHERE Ssn = :OLD.MgrSsn;
    UPDATE Employee SET Role = 'w' WHERE Ssn = :OLD.MgrSsn
       AND Ssn IN (SELECT EmpSsn FROM works_on);
    UPDATE Employee SET Role = 'l' WHERE Ssn = :OLD.MgrSsn
       AND Ssn IN (SELECT LeaderSsn FROM Project);
  END IF;
END;

CREATE TRIGGER handle_dept_upd
  AFTER UPDATE ON Department
  FOR EACH ROW
BEGIN
  IF :OLD.MgrSsn IS NULL and :NEW.MgsSsn IS NOT NULL
   OR :OLD.MgrSsn IS NOT NULL and :NEW.MgsSsn IS NULL
   OR :OLD.MgrSsn <> :NEW.MgsSsn THEN
    UPDATE Employee SET Role = 'a' WHERE Ssn = :OLD.MgrSsn;
    UPDATE Employee SET Role = 'w' WHERE Ssn = :OLD.MgrSsn
       AND Ssn IN (SELECT EmpSsn FROM works_on);
    UPDATE Employee SET Role = 'l' WHERE Ssn = :OLD.MgrSsn
       AND Ssn IN (SELECT LeaderSsn FROM Project);
    UPDATE Employee SET Role = 'm' WHERE Ssn = :NEW.MgrSsn;
  END IF;
END;

CREATE VIEW DeptManager
(Ssn,Role,Birthdate,Sex,Firstname1,Firstname2,Firstname3,
 Lastname,Title,Address,Skill1,Skill2,Skill3,DeptName,Salary)
AS SELECT Ssn, Role, Birthdate, Sex, Firstname1, Firstname2,
          Firstname3, Lastname, Title,, Address,
          Skill1, Skill2, Skill3, DeptName, Salary
     FROM Employee WHERE Role = 'm'
WITH CHECK OPTION;

CREATE VIEW ProjLeader
(Ssn,Role,Birthdate,Sex,Firstname1,Firstname2,Firstname3,
 Lastname,Title,Address,Skill1,Skill2,Skill3,DeptName,Salary)
AS SELECT Ssn, Role, Birthdate, Sex, Firstname1, Firstname2,
          Firstname3, Lastname, Title,, Address,
          Skill1, Skill2, Skill3, DeptName, Salary
     FROM Employee
    WHERE Role = 'l' OR Ssn IN (SELECT LeaderSsn FROM Project);

CREATE VIEW ProjWorker
(Ssn,Role,Birthdate,Sex,Firstname1,Firstname2,Firstname3,
 Lastname,Title,Address,Skill1,Skill2,Skill3,DeptName,Salary)
AS SELECT Ssn, Role, Birthdate, Sex, Firstname1, Firstname2,
          Firstname3, Lastname, Title,, Address,
          Skill1, Skill2, Skill3, DeptName, Salary
     FROM Employee
    WHERE Role = 'w' OR Ssn IN (SELECT EmpSsn FROM works_on);

CREATE VIEW Secretary
(Ssn,Role,Birthdate,Sex,Firstname1,Firstname2,Firstname3,
 Lastname,Title,Address,Skill1,Skill2,Skill3,DeptName,Salary)
AS SELECT Ssn, Role, Birthdate, Sex, Firstname1, Firstname2,
          Firstname3, Lastname, Title,, Address,
          Skill1, Skill2, Skill3, DeptName, Salary
     FROM Employee WHERE Role = 's'
WITH CHECK OPTION;

CREATE VIEW Assistant
(Ssn,Role,Birthdate,Sex,Firstname1,Firstname2,Firstname3,
 Lastname,Title,Address,Skill1,Skill2,Skill3,DeptName,Salary)
AS SELECT Ssn, Role, Birthdate, Sex, Firstname1, Firstname2,
          Firstname3, Lastname, Title,, Address,
          Skill1, Skill2, Skill3, DeptName, Salary
     FROM Employee WHERE Role = 'a'
WITH CHECK OPTION;
```

Figure 2.10: Definition of the Tables, Keys, Foreign-Keys, Indices, Triggers, and Views for the Company Database.

```
insert into Employee
 (Ssn,Role,Birthdate,Sex,Firstname1,Firstname2,Firstname3,
  Lastname,Title,Address,Skill1,Skill2,Skill3,DeptName,Salary)
values
 ('123456789','m','1996/04/23','m','David',NULL,NULL,
  'Miller','Dr.',NULL,'Computers',NULL,NULL,'Computer Science','117000');
```

However, the following would be rejected by the DBMS

```
insert into DeptManager
 (Ssn,Role,Birthdate,Sex,Firstname1,Lastname,Title,Skill1,DeptName,Salary)
values
 ('123456789','w','1996/04/23','m','David','Miller','Dr.',
  'Computers','Computer Science','117000');
```

because $DeptManager$ is restricted have the role 'm', but this statement uses 'w' as value for the *Role* attribute.[3]

Updating the *Department*'s *Dnumber* or $Name$ is forbidden by the REVOKE statement. But, since it is not forbidden to update the $MgrSsn$, $MgrStartDate$, and $SuperDept$ attribute of the *Department* relation, David Miller can subsequently become really the manager of the Computer Science department:

```
update Department set MgrSsn = '123456789' where Dnumber = '11.3.1';
```

2.3.5.2 Properties of some Special RDBMSs

Relational DBMSs (RDBMSs) of today, e.g. Ingres, Informix, Oracle, or Sybase, use data storage techniques based on ISAM and Btree organization. We will consider these in the Cost Model Section of the RADD/raddstar Conceptual Database Design Optimizer (Chapter 5). However, some important properties of relational database and database application implementations which are essential for the behavior considerations and transaction cost evaluations, such that we want to mention them already here, are:

1. Flexible data types. RDBMSs of today support more flexible data types than those originally considered by the relational data model. These are, for instance, variable-length character strings (*varchar*), unlimited text strings (*text*), or large objects (binary large objects $BLOBs$, or character large objects $CLOBs$). $LOBs$ allow to store individual forms of data by the database, such as scientific data or images.
2. SQL interfaces. In the past, often third-generation-language (3GL) programs were used. These base on a programming language, such as C, and embed SQL statements by means of special directives. Then a preprocessor had to be run on that embedded SQL programs to produce the pure code which is understandable to the (C-)

[3] Unfortunately, traditional RDBMSs, such as Informix, Ingres, or Oracle7, do not support *view inserts,* such that when issuing *insert into* $DeptManager$ the *Role* value– which is always 'm' –could be omitted.

compiler. RDBMSs of today provide higher level fourth-generation-language (4GL) interfaces whose language includes SQL and procedural programming language elements, such as declaration of variables and cursors, or use of loops. Dynamic SQL which is applicable in 3GL and 4GL programs allows to code SQL statements into strings which are dynamically constructed, and to PREPARE and EXECUTE them. This way, applications which support functionalities that are normally reserved to the database administration tools, e.g. *creatdb* or $SQLDBA$, can be created.

3. Locks.

   (a) General lock handling. Some RDBMSs handle transactions such that the one who issues the database retrievals sets the access permissions for himself. E.g., an Ingres user must specify if he wants to read only data that are in committed state, or he wants to do "dirty reads" as well, by the SET LOCKMODE SESSION command. The user of Informix or Oracle who makes modifications to the data, sets the authorizations whether other users have read access to the data which are currently modified by him– by specifying "exclusive" and "shared" locks with the help of the LOCK TABLE command. The user of Oracle also has the possibility to establish transactions as *read-write* or *read-only*, and to determins (by the *isolation level*) whether as transaction issuing a DML statement fails (*serializable*) or waits (*read committed*) when wanting to read data that are locked by another, uncommitted transaction.

   (b) Granularity of locks. Locks can be set on sets of relations, that the user wants to modify (or read), single relations, or single records. Options for relation and record locking are specified by the *create table* or *alter table* commands, e.g. "LOCK MODE IS ROW".

   (c) Locks on data that are read by the transaction herself. It is further possible that the DBMS allows or allows not to change the content of a table that is currently read– even if it is read by the same user who makes the modification. This can usually be configured by the startup files for the database server processes, so using Oracle by the *initDB.ora* file. This file defines buffer sizes for undoing transactions and the concurrent execution of database modifications *(shared_pool_size)* and for the maximum number of operations which include retrievals *(open_cursors)*. Besides select and special cursor operations that are specified by means of a select statement, insert operations which include a check for foreign-key values, e.g. the insert operation on the *Employee* relation which ensures that the value of $DeptName$ is contained as $Name$ in *Department*, are using such cursors. Therefore, having defined the *open_cursors* value high enough in the *initDB.ora* file, it is possible to have an application interface which has a range of drop-down lists that are filled using database selects, and

to issue the command

```
delete from Department
 where SuperDept is not null
   and not Dnumber in (select SuperDept from Department)
   and not Name in (select DeptName from Employee)
   and not Dnumber in (select ContrDept from Project);
```

to delete all Departments which have not anymore employees and projects, and are also not supervising other Departments. The standard parameters of Informix, on the other hand, report such a command with a lock conflict ("NO MORE LOCKS AVAILABLE"), for reason that the Department relation is contained in the first sub-select.

(d) Two-phase locking (2PL). The relational DBMSs mentioned here support *two-phase locking* and the *two-phase commit protocol*. That is, if two transactions are running concurrently and are sharing some records (or tables) which they read and/or modify, and a transaction waits for a record that is locked by the other transaction for a short time, then the transaction that wants to commit sends a signal *ready to commit* to the DBMSs. Only if both transactions were successful, the DBMSs registers their commited results by a new database state, otherwise it rejects their modifications. This avoids that transactions may work with database records which are immediately locked and perhaps modified by other transactions that do no commit.

(e) Savepoints. Some DBMSs, e.g. Ingres, additionally support *savepoints* which can be set during the operation sequences of transactions, such that it is possible to issue partial transaction rollbacks (which are performed by *rollback to savepoint* commands), and to restart operating using the database state at the savepoint.

4. Further reading. These RDBMS functionalities are defined in the SQL-92 documents [X3.92], and can also be looked at more closely by inspecting special DBMS manuals, such as [Syb93] or [PLSQL95].

2.4 Summary and Outlook

The data models and database management concepts presented in this Chapter provide the design and maintenance of data for traditional database applications. That is, data which normally can be represented in the form of flat tables, and, integrity constraints which do not need to be controlled by active elements, but can be implemented by the the database structure.

But as we have seen by the examples of the database implementation code, today's database applications sometimes use unclean representations of data, since the correct

representation can not be represented in a consistent way by the concepts provided by the used data model and DBMS. As examples, consider

- the attribute *DeptName* of the *Employee* type in Figure 2.7 and 2.8, or the repeating groups in Figure 2.10 which are represented by the attributes $Firstname1$, $Firstname2$, $Firstname3$ and $Skill1$, $Skill2$, $Skill3$,
- or, the trigger definitions shown in Figure 2.10, which are used to confirm that $Employee.DeptName$ is in $Department.Name$, and to assign an employee his according role in the Company ($DeptManager$, $ProjLeader$, $ProjWorker$, $Secretary$, or $Assistant$).

Integrity control can already be specified during design and not only after the database is installed. Therefore, new-generation database concepts need to consider any form of structured data as well as active integrity control mechanisms, for purpose to repair incorrect and incomplete structural database design decisions immediately.

We will consider these new requirements to data models and database management systems in Chapter 3.

Chapter 3

New-Generation Database Design and Database Management Approaches

In Chapter 2 we have presented traditional database design and management concepts. The network model already included concepts that were omitted by the relational data model: set-typed attributes and one-to-many relationships which are contained in the same record sets. Also, in traditional database systems, new database states are generated only by applications which are again maintained by the users, and are invoking the insert, delete, or update operations. But, new-generation database applications like *computer integrated manufacting* (CIM), *computer aided design* (CAD), or Web database interfaces require the additional consideration of temporal relations, which– once they are introduced into the database– cause the database management system to generate new database states by means of invoking additional actions.

In this Chapter we will consider the new structural requirements as well as the requirements to perform database maintenance tasks automatically:

1. Attribute types for structured and collection-typed data, as well as attribute types which can be defined by the user.
2. Encapsulation, which is the linkage of the data structures with exectable code.
3. Inheritance, which gives objects their extensibility. New data structures ("classes") that are derived from anothers inherit all properties of that other class, and have also the properties that are added to it.[1]
4. Polymorphism, which we have not considered till now.
5. Database triggers, which are used to trigger additional actions as soon as certain events occur, such as database changes which are the result of an insert, delete, or

[1]The "is_a" relation between "Employee" and "Manager" that was shown in Figure 2.4, describes some limited form of inheritance. Although a very weak kind, multiple inheritance was also given by the many-to-many relationship types of Figure 2.1.

update operation, or of a complex user transaction.

6. New concepts of graphical user interfaces, such as HTML pages which are provided by the Web server to perform retrievals and changes on a database.

Chapter 3 is organized as follows. Firstly, Section 3.1 considers functional and semantic data models, which are often looked at as the origin of developing new-generation data representation and database management concepts. This is continued with the approach to database design, that is used by object-oriented database design methodologies in Section 3.2. Section 3.3 presents additional enhanced data modeling concepts, which are refered to in the forthcoming Chapters. Section 3.4 summarizes the data models discussed in this Chapter, and gives an outlook how these data models are considered by the RADD/raddstar conceptual database design optimizer.

3.1 Functional and Semantic Data Models

Functional data models have once been developed to give foundation for the implementation of persistent programming languages which are retaining data after the program is ended, such that they can be used again when the program is newly started. Semantic data models were considered already in the seventies, as a reaction to the simplicity of the relational data model. In these models, any semantics that could not be modeled by means of relations, is embedded in application code.

Although the semantic and functional data models have never gained interest using them as basis for implementing new-generation object-oriented DBMSs in the ending 80's and beginning 90's, their concepts had strong influences on the developmemt of these systems. So, some ODBMSs borrowed a range of concepts from these models. In this Section we will consider three of these data models. The first is the functional data model and the language DAPLEX developed by Shipman [Shi81]. Then we condider the semantic data model (SDM) of Hammer and McLeod [HM81]. The last model considered in this Section is the IFO data model of Abiteboul and Hull [AH87], which was developed from the functional data model and SDM, and is today frequently used for modeling object database systems.

3.1.1 The Functional Data Model and the DAPLEX Language

The functional data model was introduced by Kershberg and Pacheco [KP76], and refined by Sibley and Kershberg [SK77]. Shipman [Shi81] uses DAPLEX as the database modeling and manipulation language to implement the functional data model. The base constructs of DAPLEX are entity and function. In the functional data model, we can consider *Project* and *Employee* as entities with the function *has_participants* to map one to the

other: a DAPLEX function is used to map one entity to a set of entities, similar the owner-member relationship types of the network model.

Real world situations are presented in the DAPLEX model as either "primitive" or "derived" properties. E.g. from the primitive property *a project has participants* (which are the participants of that project), we could derive the property that *one of these participants leads the project* (is the project leader). Furthermore, the functional data model is based, as its name says, on the concepts of handling functions and function evaluations as values, like functional programming languages do. This way, queries and updates can be expressed by including and combining query results as well as navigation paths in their formulation. They have not necessarily to be implemented in a step after step manner like procedural programming lanuages require, such that new declarations and assignments are continously used.

Example of a DAPLEX query. Assume, a department of the company has allocated a new project for that it needs an employee with skill "Mathematics", who is not still available. Then it may look for a project which has some participant with skill "Mathematics", and is able to release this participant for the new project. For this purpose, to find out the project(s) from which the department can get a participant, the following query

Which are the projects on which an employee with skill "Mathematics" works?

can be expressed in DAPLEX as

```
FOR EACH Project
  SUCH THAT FOR SOME has_participants(Project)
    SUCH THAT FOR SOME Employee(has_participants)
      Name(Skill(has(Employee))) = "Mathematics"
  PRINT Name(Project)
```

Like in the hierarchical model and network model, a query or update in the functional model as well as a functional data schema modeling the static aspects of the mini world, is represented by a directed graph.
In the graphical representation of a DAPLEX functional data model, the functions are represented by arrows with a single angle (for functions which evaluate to exactly one value) or by arrows with two angles (for functions which can evaluate to more than one value[2]). The arrows which are connecting attributes, are drawn by arrows with a single angle. In Figure 3.1 we show the functional data model of the Company Schema, in which we have, for reason of better understandability, not included all attributes that were shown in the entity-relationship schema of Figure 2.1.

[2]Arrows with two angles can be considered similar the multivalued dependencies of the relational model (Section 2.3.3.1). Multivalued dependencies for relational databases were introduced by Fagin [Fag77], long after they were considered by the network model and the functional data model.

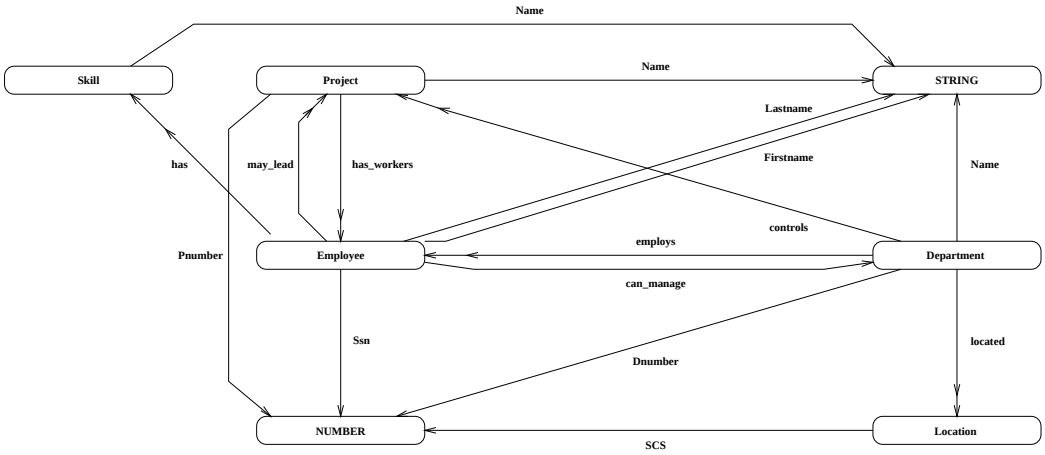


Figure 3.1: Functional Data Model of the Company Database.

3.1.2 The Semantic Data Model (SDM)

The semantic data model was proposed by Hammer and McLeod [HM81] considering that the Codasyl proposal for database management and the hierarchical data model *exhibit compromises between the desire to provide a user-oriented database organization and the desire to provide efficient storage and manipulation facilities*, and that *the relational database model stresses the separation of user-level database specifications and underlying implementation detail (data independence).*

The SDM therefore presupposes that (1.) a database must be viewed as collection of entities, (2.) the collection of these entities must be viewed as classes, (3.) the classes are not independent, but logically connected, (4.) the classes and the whole database are described by attributes mapping their characteristics, such that there are attributes whose values can be derived from others, and (5.) there are several ways of defining interclass connections and deriving attribute values from values of other attributes, which are depending on the *most common types of information redundancy in database applications.*

This way, basic concepts of the SDM are:

1. *classes,* which are mapping collections of entities, and are distinguished into *base classes* whose instances are living independently in the real world, like entity types in the ERM, and *nonbase classes,*
2. *interclass connections,* which are *subclass connections* like "is-a" relationship types in the ERM, *grouping connections* (binary relationship types in the ERM), and *multiple interclass connections* (relationship types with arity >2 in the ERM),
3. *attributes,* which describe the properties of the classes and the interclass connections, and
4. *name classes,* which can be considered as atomar domains, such as integers, strings, etc., and are used as buildings blocks for the classes and the attributes.

To illustrate the data presentation of the SDM, let us take a look at Figure 3.2.

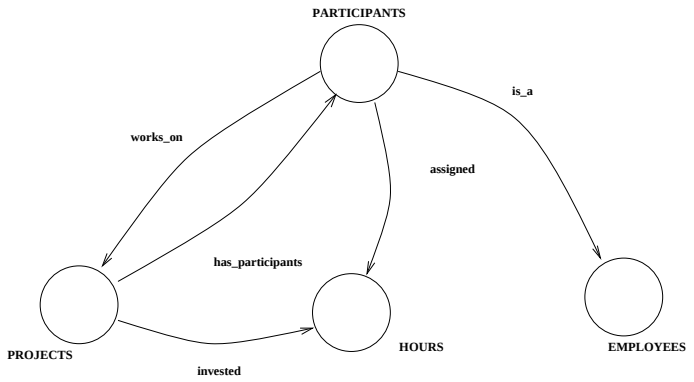


Figure 3.2: Multiple properties of the "PARTICIPANTS" subclass. The circles denote classes and are labeled with the class names. The arrows which are labeled by a name denote member attributes, with the arrow head (angle) pointing to the attribute's value class. For transparency, only some of the possible attributes are included here.

Figure 3.2 models the *PARTICIPANT* view of an *EMPLOYEE* who *works on* a *PROJECT*. For the participation on the *PROJECT*, *HOURS* are assigned to the *EMPLOYEES*, which are *invested* by them into the *PROJECTS*. The *PROJECT* sees the *PARTICIPANTS* as its workers (*has_participants*), and the *EMPLOYEES* may be *PARTICIPANTS* of the *PROJECTS*.

Since the SDM uses no graphical description *per se*, like that shown in Figure 3.2, it defines a formal description language to model the mini world. So the mini world of Figure 3.2 is formally specified by the following SDM description (also called SDM schema), where the attributes of *EMPLOYEES* and of *PROJECTS* are included:

```
EMPLOYEES
  description all people who are working for departments of the company
  class attributes
    Ssn
      value class PERSON__SSNS
      may not be null
      not changeable
    Name
      description
        the name of an employee consists of an (ordered) set of firstnames
        and a lastname, and may include a title
      valus class PERSON__NAMES
      may not be null
    Birthdate
      value class DATES
      may not be null
    Address
      value class STRINGS
    Sex
      value class SEXES
      may not be null
    Salary
      value class INTEGERS where >=10000
  identifiers
    Ssn
    Name
PROJECTS
```

```
  description projects which are acquired and performed by the company
  member attributes
    has_participants
      value class PARTICIPANTS
      multivalued
    invested
      value class HOURS
      multivalued
  class attributes
    Pnumber
      value class PROJECT__NUMBERS
      may not be null
      not changeable
    Name
      value class STRINGS
      may not be null
    Location
      description the location where project grew works (free text)
      value class STRINGS
      may not be null
    ProjStartDate
      description the estimated start date of the project
      value class DATES
    Duration
      description the estimated duration of project in months
      value class INTEGERS where >=1
  identifiers
    Pnumber
HOURS
  description the hours which are assigned to an employee to work on a project
    value class INTEGERS
PARTICIPANTS
  description all employees which are participants on a project
  member attributes
    is_a
      value class EMPLOYEES
      may not be null
    works_on
      value class PROJECTS
      multivalued
    assigned
      value class HOURS
  identifiers
    is_a
PERSON__SSNS
  interclass connection subclass of STRINGS where format is number where integer
PERSON__NAMES
  Firstnames
    value class STRINGS
    may not be null
    multivalued
  Lastname
    value class STRINGS
    may not be null
  Title
    value class STRINGS
SEXES
  interclass connection subclass of STRINGS where format is "male" or "female"
DATES
  description calendar dates in the range "1/1/1998" to "12/31/2005"
  interclass connection subclass of STRINGS where format is
    month number where >=1 and <=12
    "/"
    day number where >=1 and <=31
    "/"
    year number where integer and >=1998 and <=2005
    where if month = 4 or = 6 or = 9 or = 11 then day <=30
      and if month = 2 then if year = 2000 or = 2004 then day <=29 else day <=28
    ordering to year, month, day
PROJECT_NUMBERS
  description numbers which given for projects of the company
  interclass connection subclass of STRINGS where format is
    year number where integer and >=1998 and <=2005
    "."
    month number where integer and >=1 and <=12
    "."
    num number where integer >=1
```

In the SDM schema, *EMPLOYEES* is a *base class*, and *PROJECTS* and *PARTICIPANTS* are *nonbase classes*. *HOURS*, *PERSON__SSNS*, *PERSON__NAMES*, *SEXES*, *DATES*, and *PROJECT__NUMBERS* are *name classes*. These as well as the classes' *attributes* which are not using *name classes* defined here, are built from predefined *name classes* (*INTEGERS*, *STRINGS*, etc.). The *name classes* defined here are built from predefined *name classes* as well, and add lower and upper bounds, or special formats, or use aggregations *(PERSON__NAMES)*. Note that *interclass connections* ("relationships") are included in the *class* specifications.

We have not included all possible features in the SDM schema, but we think that it is enough to give the reader an impression of the specification techniques of SDM.

3.1.3 The IFO Database Model

The IFO data model was proposed by Abiteboul and Hull [AH87] as a formal semantics database model. It provides the graphical notation of the database's structural component and the specification of data manipulation, and claims to specify the integrity component as well. Furthermore, it tries to give lead to the hierarchical construction of database schemata, which the authors presuppose as a necessary condition for *a rigorous, mathematical investigation of semantic database issues.*

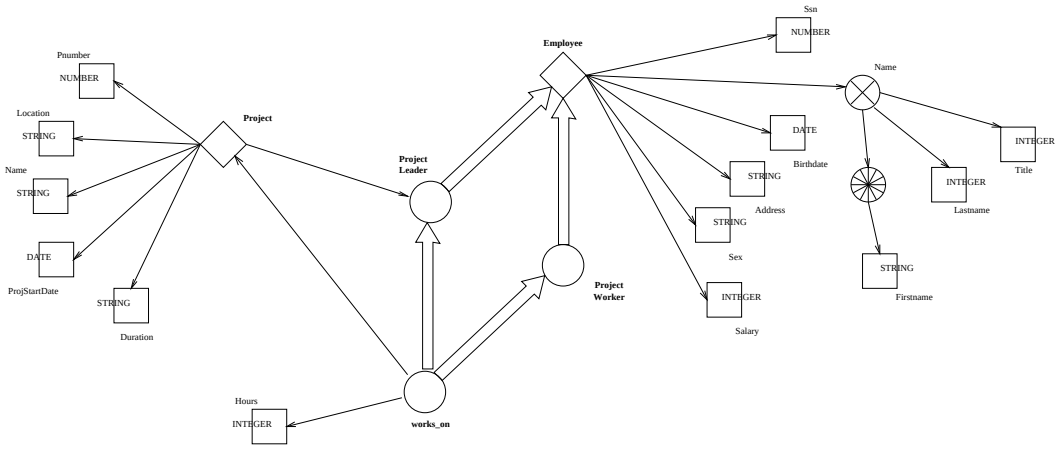


Figure 3.3: IFO Schema of the Employee/ProjWorker-works_on-Project/Project-Leader-leads-Project Section of the Company Schema.

Basic IFO data model constructs are (1.) objects and object-types, (2.) fragments, and (3.) ISA-relationships. For illustration purpose, let us consider these concepts using the

diagram in Figure 3.3, which contains the SDM schema specified in Section 3.1.2 and adds the *Employee*'s role *Project Leader*:[3]

Object types are collections of objects having the same characteristics and correspond to classes in the SDM. IFO considers three kinds of object types: *printable object types*, *abstract object types*, and *free object types*.

1. *Printable object types* are collections of values having predefined types, such as integer, string, etc. They are used as basis for input and output between objects, and correspond to value sets in the ERM and relational model, to *name classes* in SDM and to *Lexical Object Types* (LOTs) in the ORM, which will discuss in Section 3.3.1. Printable object types are graphically represented by squares and may be annotated by their types. In Figure 3.3 we see the printable object types *Pnumber*, *Name*, etc., which where attributes in the ER-Schema (Figure 2.1) and name classes in the SDM schema (Section 3.1.2).
2. *Abstract object types* are representing objects of the mini world living independently from others, like entity types in the ERM, base classes in the SDM, and *Non-Lexical Object Types* (NOLOTs) in the ORM. Abstract object types are graphically represented by diamonds. Figure 3.3 contains the abstract object types *Employee* and *Project*.
3. *Free object types* represent entities which are subtypes of ISA relationships. Free object types are graphically represented by circles. Figure 3.3 contains the free object types *Project Leader*, *Project Worker*, and *works_on*.

Cartesian products which build n-tuples from object types. In the graphical IFO notation cartesian products are represented by crossed circles. In Figure 3.3 we see the cartesian product *Name* which combines the *Employee*'s *Firstnames*, *Lastname*, and *Title*.

Groupings *which correspond to the procedure of forming finite sets of objects of a given structure type.* Groupings are graphically represented by multiply crossed circles. In Figure 3.3 we see the grouping of *Firstname*, that builds the multivalued attribute which is part of the *Employee*'s *Name* (*Firstnames*).

Specialization. To represent specialization, IFO uses ISA inheritance relationship types. In the graphical representation, ISA relationship types are drawn by wide unfilled arrows. Figure 3.3 contains the ISA relationship types between the subclasses *Project Leader* and *Project Worker*, and the superclass *Employee*. Additionally, we see the ISA relationship types between the subclass *works_on* and the superclasses

[3] The description is partly taken over from Hanna [Han95].

Project Leader and *Project Worker*. This way, *single inheritance* as well as *multiple inheritance* are supported.

Generalization. Besides specialization (ISA relationship types), IFO proposes generalization as a second kind of inheritance. Generalization is used to model situations when distinct preexisting classes are used to form a new virtual type.[4] Generalizations are graphically represented by wide filled arrows. However, [AH87] defines generalization such that the subtype shall pass its attributes to the supertype, which is semantically incorrect, and omits a clear proposal to which inheritance type the constraint types, "covers" and "disjoint", that are given according the subclasses may overlap or not, should be mapped, such that it leaves it unclear when to use specialization and when to use generalization. For reason of this unclear concept, we do not present IFO generalizations here.

Fragments are used by the IFO the same way they are used to represent functions in the FDM of Shipman (Section 3.1.1), but are restricted to model one-to-one associations between two objects. In Figure 3.4 we have represented an IFO fragment, named

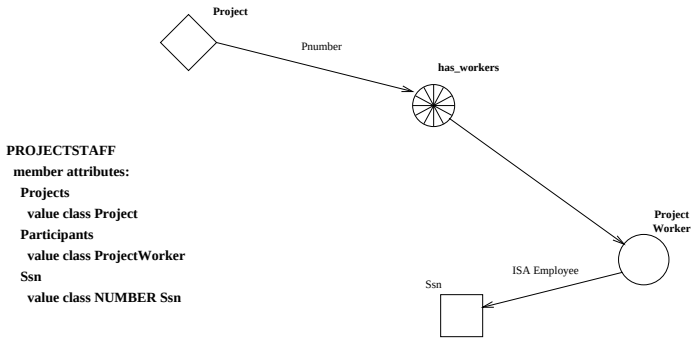


Figure 3.4: IFO Fragment "PROJECTSTAFF".

PROJECTSTAFF. The fragment shows the path for the functional evaluation by which a *Project* can determine his *participants*.

However, IFO does not introduce new semantic concepts that were not already considered by the FDM (Section 3.1.1) or the SDM (Section 3.1.2), but makes rather the notation of database semantics clumsy and ambigous, because it provides many concepts for representing the same aspects while it omits concepts for representing other aspects. E.g., IFO omits many-to-many relationship types, or does not use a notation to represent cardinality constraints.

[4] A virtual type is presented only by the instances of the preexsting classes, but has no own physical attributes which carry values.

3.2 Object Models

Object models claim to give a natural view to the real world, by modeling data containers ("classes") as well as the operations which are performed by the objects of the classes, or are applied to them ("methods").

In this Section we present some object data models which are popular and frequently used for implementation under object-oriented programming languages, such as C++, and for object modeling. The Section shows how different the approaches to object modeling are, and also, how different their consideration of concepts is, such that they are the less or the more suitable for database modeling.

```
// dates
typedef struct { short day, month, year; } Date;

// the list of Firstnames
typedef struct SFns { char Firstname[15]; struct SFns *next; } *Firstnames;

typedef struc { Firstnames firstnames; char lastname[20]; char *title; } Name;

// the set of Skills
typedef struct SSkls { char skill[20]; struct SSkls *left, *right; } *Skills;

class Employee {
public:
  Name *getname();
  Firstnames firstnames(Name *name);
  char *lastname(Name *name);
  char *title(Name *name);
  ...
private:
  char Ssn[9];
  Name Name;
  Date Birthdate;
  char Sex;
  ...
  Employee(char *ssn,char **fns,char *ln,char *tit,char sex,Date bd,float sal,char *dept,char* ad,Skills skls);
  ~Employee();
};

Employee::Employee(char *ssn,char **fns,char *ln,char *tit,char sex,Date bd,float sal,char *dept,char* ad,Skills skls)
{
  strcpy(Ssn,ssn);
  strcpy(Name.lastname,ln);
  int i; Firstnames h;
  for(Name.firstnames=NULL,i=0; fns && fns[i]; i++)
   { new(h); strcpy(h->firstname,fns[i]); h->next=NULL; if(!i) Name.firstnames=h; h=h->next; }
  Name.title=(tit) ? strdup(tit) : NULL;
  ...
};
```

Figure 3.5: C++ Definition/Implementation of the Employee Class.

3.2.1 The Booch Method

The Booch method [Boo94] considers that the real world is too complex to understand it as a whole such that a good approach is to decompose it into smaller units. This way decomposition, abstraction, and hierarchy play important roles in Booch's approach to object-oriented analysis and design.

The Booch method supports single and multiple inheritance; "is a" hierarchies are considered as the most important hierarchy types and as *an essential element of object-oriented systems*. They describe generalization/specialization. On the other hand, "part of" hierarchies describe aggregation relationships.

Like in other object models, the basic concepts of the Booch method are classes and objects; refering to these parts we could consider the *Employee* entity-type of Figure 2.1 as a class that is in C++ represented with external interfaces ("public" methods) and an internal representation ("private" attributes), shown in Figure 3.5. Here, for simplicity we did not make use of "protected" attributes and methods, that are seen by objects of the subclasses only, and we did not show "virtual" methods which can be overridden by subclasses of *Employee*.

Beyond the concepts of class structuring that it defines and illustrates and object-oriented programming of it in C++, the Booch method does not define a graphical notation for the representation of classes, objects, and their interrelationships. We therefore break at this point, in order to continue with object models which possess a graphical notation and can be used for database design as well.

3.2.2 The Object Modeling Technique (OMT)

The object modeling technique (OMT) has been developed at General Electric by James Rumbaugh and colleagues, and was published in 1991 [RBP+91]. It is based on traditional structured methods and offers a rich and detailed notation, which is sometimes a little bit unreadable for the one who is not primarly and continously concerned with data modeling. The OMT notation has roots in the entity-relationship model and adds operations and other annotations to it.

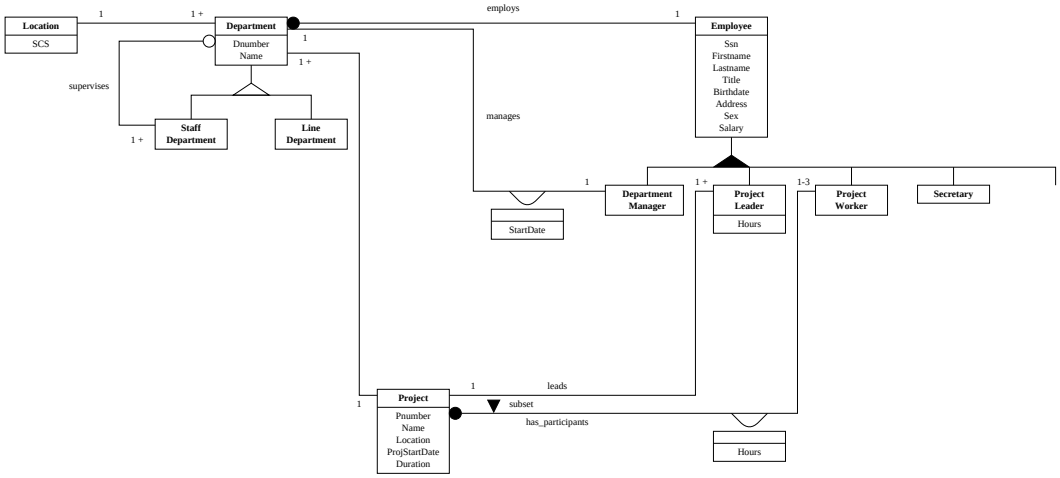


Figure 3.6: OMT Object Model of the Company Database.

Object modeling with OMT is based on three modeling steps. The first step is to develope the object model (OM) of the mini world that is under consideration, next for each object a dynamic model (DM) is built, and finally, flows of data are modeled by means of data

flow diagrams, which are called functional model (FM) in OMT.

The diagram in Figure 3.6 shows how the Company Schema is represented as object model (OM) in OMT. In the OM the lines between the boxes (classes) represent the associations between the object (relationship types). The associations can have a name which is drawn near by the line. As in the ERM, the associations can have attributes, which are attached by means of a semicircle arc to the association. The OM uses triangles for generalizations/specializations ("is_a" relationship types) which are empty if the objects of the superclass are either one of the objects of the subclasses ("distinct specialization"). The diagram in Figure 3.6 defines the objects of the superclass *Department* as belonging either to $Staff\ Department$ or to *Line Department*. An attribute called "discriminator" which is annotated to the triangle (not shown here) can also be used to differentiate to which of the subclasses the object of the superclass belongs. An other example, of a "non-distinct specialization", is represented by the filled triangle between the superclass *Employee* and her subclasses $DeptManager$, $ProjLeader$, $ProjWorker$, *Secretary*, and possibly further subclasses which must not be explicitely specified.

Associations can also be ternary, quarternary, and so forth, which we have not (already) included in the Company Schema and are therefore omitted here. We have represented the classes including their attributes (*Dnumber*, *Name*, etc.) which is not necessary, but better to provide a complete design. The unfilled circle at the *Department* head of the line mapping the *supervises* association denotes a zero-or-one cardinality– it were $(0, 1)$ using the notation of Section 2.3.4.1. If such a circle is filled, like the one at the *Project* head of the *has_participants* association, then it denotes a zero-or-one-to-many cardinality such that the lower and the upper bound are left open– it were $(0, .)$ using the notation of Section 2.3.4.1. Other types of cardinality constraints, like $(1, 1)$ or $(1, 3)$, are explicitely drawn by means of the according annotations, e.g. 1 or 1-3.

The database schema of this Section shows another aspect which we have not considered till now, namly that the one who *leads* the project ($ProjLeader$) must work on that project. In Figure 3.6 we have modeled this by the *includes* triangle between the *leads* and the *has participants* (*works_on*) association. This gives the diagram a clear non-misunderstandable meaning– considering the database implementation that is derived from the OM.

However, although the OMT provides a rich and expressive notation framework it confuses a little bit by using different notation kinds; the cardinality notation is an example for such a non-unique modeling tool that is included by OMT. For example, the cardinality constraints represented by the filled and unfilled circles of the *employs*, *has_participants*, and *supervises* associations, headed at *Department*, *Project*, and *Department*, respectively, can be expressed by 0-m or 0-1 annotations as well. This would make them conform with the other cardinality annotations.

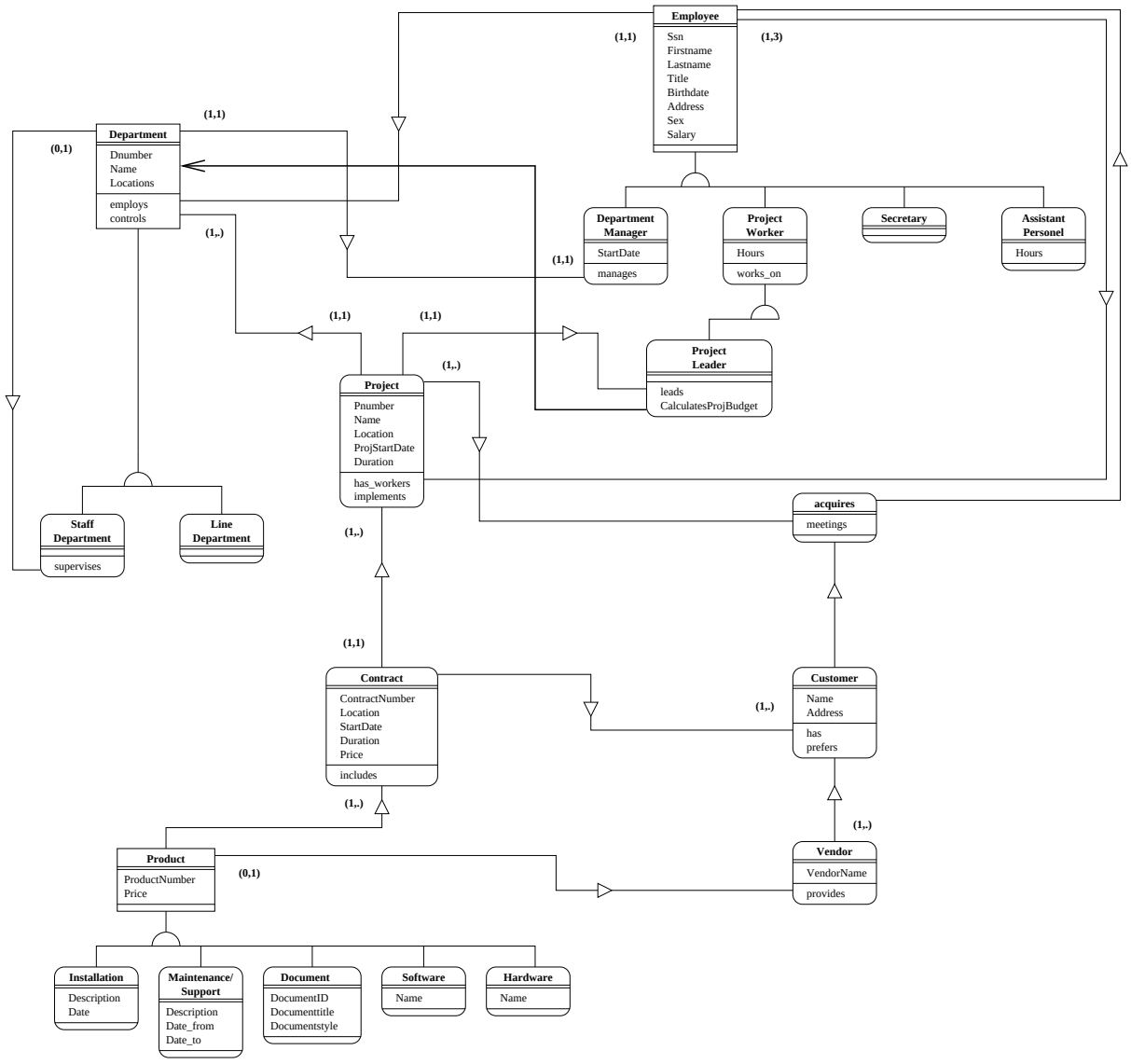


Figure 3.7: Coad/Yourdon Model of the Company Database.

3.2.3 The Coad/Yourdon Method

Like the OMT, the Coad/Yourdon method owes a lot of the tradition of entity-relationship models. E.g., the use of cardinality notation and the expliciteness of attributes are some of these features. The Coad/Yourdon method can be used for object-oriented analysis (OOA) as well as object-oriented design (OOD). It includes the following concepts:

Classes, which represent collections of objects with the same properties. In the Coad/Yourdon method, *abstract classes* are distinguished from *concrete classes*. Abstract classes (which are drawn by solid boxes in the graphical model) form abstractions of objects that are living in the real world, that have no concrete instances– like *virtual classes* in C++ –, whereas concrete classes (arc boxes) represent concrete objects. A class is specified by her class name, her attributes, and her services, like a "Department employs Employees" such that a service "employs" is attached to the "Department" class.[5] The attributes of the abstract classes as well as of the concrete classes describe their properties, and the services (methods) describe their behavior.

Attributes, which represent properties of the objects, such as the "name" of a "Department", the same way like attributes in the relational model and the OMT.

Services, which are a synonym for functions or *methods*, in other data and object models. In contrast to the attributes that are properties of the objects (instances), services are considered to be connected to classes, like in object-oriented programming languages.

Gen/Spec connectors, which– like in the OMT –model the specialization of superclasses to their more concrete instances, which are represented by the subclasses. Superclasses and subclasses are normally not distinguished graphically in the Coad/Yourdon method, besides that the superclass is drawn above the Gen/Spec symbol and the subclasses are drawn below the Gen/Spec symbol. However, in most cases the superclass is modeled by an abstract class which is assumed to have no concrete instances, such that it is represented only by the attribute values of her subclasses, and the subclasses are modeled by concrete classes. Consider the *Employee* class in Figure 3.7, which we have represented as an abstract class. That Coad/Yourdon diagram proposes, whenever using an SQL database implementation, to implement *Employee* by a view that unites the classes *Department Manager*, *Project Leader*, *Project Worker*, *Secretary*, and *Assistant Personel*, which is different from that representation we have used in the create statements of Figure 2.10.

[5] This make not be seen as intent of the Coad/Yourdon method, but since Coad/Yourdon does not attach names to the connectors (associations, or relationship types), we have encoded these names here by services carrying the name of the association.

Whole/Part connectors, are types of relationships between structures such that the objects of the Whole-side of the connector are considered as possessor (owner) of the objects of the Part-side of the connector.

Instance Connectors, which can be compared to many-to-many relationship types in the ERM. But, in contrast to the many-to-many relationship types of the ERM (and OMT) many-to-many connectors in Coad/Yourdon can not have attributes. Therefore, if he wants to represent a many-to-many relationship type with attributes the designer has to use a separate class for the association, which has the attributes and is many-to-one connected to each of the classes of the original many-to-many relationship type. Accordingly, Whole/Part connectors of the Coad/Yourdon method can also not have attributes.

Cardinality Constraints are used in Coad/Yourdon diagrams as well. For example, in Figure 3.7 the cardinality on the *Employee* head of the Whole/Part connection between *Project* and *Employee* (*has_workers*), which is $(1, 3)$, specifies that every *Employee* works on at least 1 *Project* and at most 3 *Projects*.[6]

Figure 3.7 shows the Coad/Yourdon model of the Company database. The diagram now includes another section of the mini world which we are considering here, representing that projects are contracted with a customer:

1. A project is acquired by some employee of the company. Most times, department managers or project leaders acquire new projects, but we do not want to exclude that any other kind of employee acquires a new project. We keep track on the meetings that where made under participation of this employee, to acquire the project.

2. Once a project is confirmed, the company makes a contract with the customer from which the project was acquired. We keep track on the customer's name and address. Furthermore, each contract is assigned a unique contract number, a location, a start date, a duration, and a price. The project implements that contract, and a contract is implemented by exactly one project. However, to continue a project additional contracts can be made which are then implemented by the same project.

3. Each contract includes products which are assigned a product number and a price within the contract. Products can be of different types: installation tasks, maintenance support, documents, hardware, or software. We keep track on the installation's description and date, on the maintenance support's description, start date

[6] We use here the same notation for cardinality constraints, as introduced in Section 2.3.4.1. Normally, in Coad/Yourdon diagrams annotations like 1-3 and 1 are used to represent cardinality contraints $(1, 3)$ and $(1, 1)$, respectively.

($Date\_from$), and ending date ($Date\_to$), on the software's name, and on the hardware's name.

4. If the product is provided from a third party, such as hardware and software which are delivered by another company, then we keep track on the vendor who provides that product. For simplicity, the vendor is stored only by his name in our database ($VendorName$). We keep track on the vendors which are prefered by the customer.

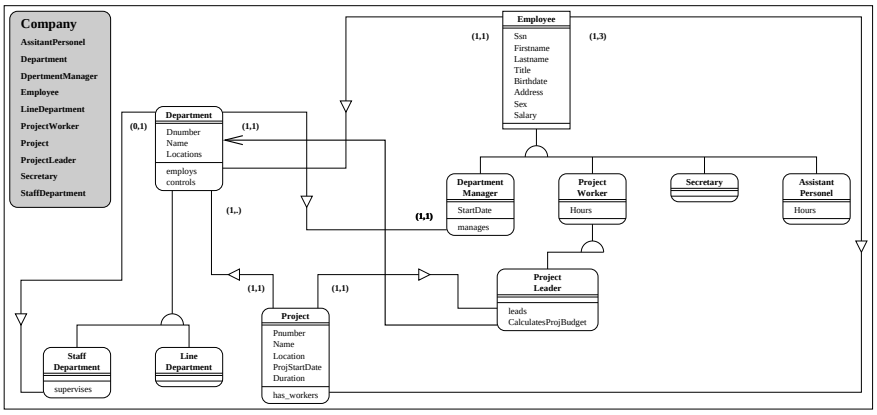


Figure 3.8: Coad/Yourdon Model: Subject "Company".

Further concepts of the Coad/Yourdon method which we have not already mentioned are:

Message Transport connectors. The modeling of messages in Coad/Yourdon is based on concepts of the *Smalltalk* programming language. In Smalltalk, a "message" which is comprised of the following information:

1. the object to which the message is send, called *receiver*;
2. the method which shall be applied to the object, called *method-selector*;
3. and the arguments which are passed to that method.

The object which invokes the method– that sends a message to the method –is considered as *sender*. In the Coad/Yourdon model, message passing of objects is specified by senders and receivers as well. These can be attached to classes and are indicated by arrows in the graphical Coad/Yourdon representation. E.g., in Figure 3.7 and 3.8 the arrow from the $ProjLeader$ class to the $Department$ class specifies $ProjLeader$ as sender and $Department$ as receiver. We have further represented this message passing functionality by the method $CalculatesProjBudget$. But, in RADD we are not considering applications which have active objects such as

graphical user interfaces (GUIs) that use *callback functions* to define the behavior of push buttons, etc., while we are designing databases for storage of passive objects. However, as we will see in Section 7.2.1 such functionalities– if necessary –could be specified in the RADD/raddstar using the conceptual specification language (CSL).

Subjects, which are considered as collections of classes belonging together, and the associations which are linking them (Gen/Spec, Whole/Part, and Instance connectors). In this sense, they model a section of the mini world containing objects which can be considered independently from objects of other sections of the whole mini world that is modeled.

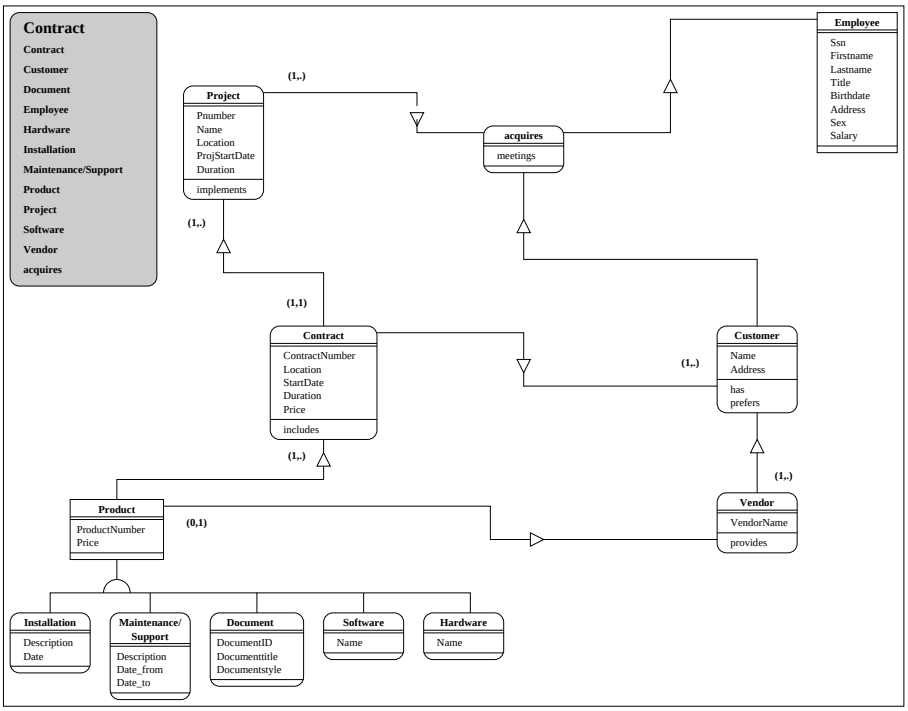


Figure 3.9: Coad/Yourdon Model: Subject "Contract".

Subjects of the Coad/Yourdon Model. As mentioned above, Coad/Yourdon models can be specialized into smaller sections, which are represented by the subjects. E.g., the model in Figure 3.7 can decomposed into two subjects, once according the working staff of the company, and once according the contracts which are implemented by the projects that are allocated by the company. Thus, we have the model also represented as subjects *Company* (which is shown in Figure 3.8) and *Contract* (which is shown in Figure 3.9).

3.2.4 Using Object Models for Database Design?

This Section did not survey on all possible object models, and did also not consider all of the "fine" features which are additionally provided by some object models. The Section did rather concentrate on three methologies which are popular and frequently used in commercial environments.

Although Booch claims to support the design of relational databases as well, his method concentrates rather on the desgin of applications that have data in main memory. So, it omits important features which are necessary for database modeling. These missing features are, for instance, integrity constraints, relational operators, or principles of database maintenance by manipulation and query languages like SQL. Beyond this, Booch does not define a notation for the graphical design, but uses C++ program examples.

The Coad/Yourdon method omits n-ary relationship types ($n > 2$), and also its representation concepts for generalization/specialization are not as complete as in OMT. But, it is simple and gives a unary notation for database designs. Both, OMT and the Coad/Yourdon method, give clear and non-misunderstandable data models for those applications which do not need or are not managed by means of complex data structures, like traditional (and current) relational databases. But, object models do not have direct support for representation of advantageous attribute structures, such as records, lists, and sets, and for integrity constraint types as usually required by database realizations, such as key and foreign-key attributes.

3.3 Enhanced Data Modeling, Database Management, and Database Specification Concepts

We have seen record- and collection-typed attributes of the network database implementation in Figure 2.7 and 2.8. In Figure 2.10 we have also shown the implementation of triggers which set the *Role* attribute of the *Employee* records correctly, after a *Department* is assigned a new manager. Therefore, enhanced database modeling concepts need to include the construction and use of types for new-generation database applications, i.e. lists, arrays, sets, records, and so forth, and to support for the informal specification of database triggers as well. In this Section, we will consider these concepts.

3.3.1 The Object-Role Model (ORM)

Nijssens Information Analysis Method (NIAM) was introduced as a kind of binary entity-relationship model (binary relationship types only) by G.M.Nijssen [Nij77]. Relationship types are called *fact types* in NIAM, and role names as well as a range of integrity con-

straints may be annotated on fact types. The *Object-Role Model* (ORM) is a recent dialect of the NIAM. Figure 3.10 shows the graphical notation concepts of the NIAM.

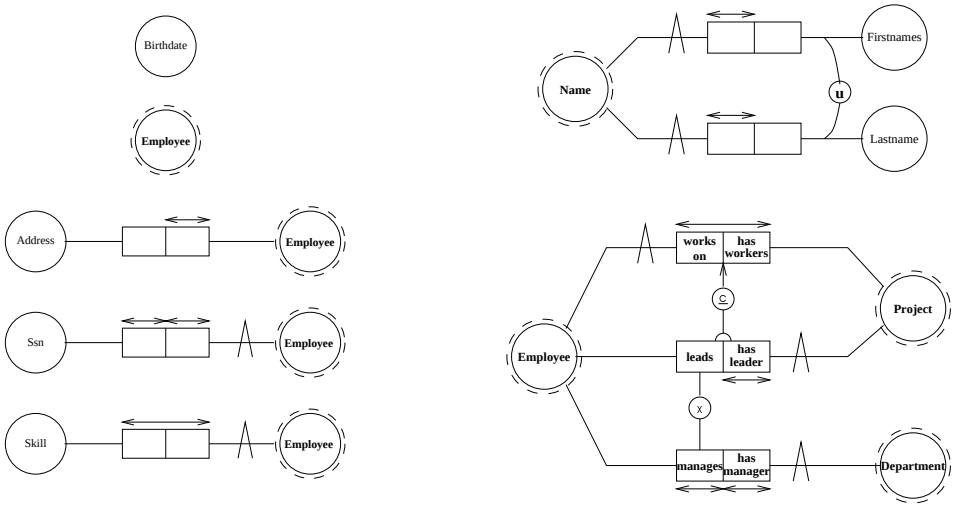


Figure 3.10: Modeling Concepts of the NIAM and Object-Role Model (ORM).

The basic concepts of NIAM are *lexical object-types* (LOTs), represented by *Birthdate*, *Ssn*, and *Skill*, and *non-lexical object-types* (NOLOTs), represented by *Employee*, *Name*, *Project*, and *Department*. LOTs can be compared to attributes, and NOLOTs to entity types of the ERM, respectively. However, ERM attributes whose values may not be described and generated in natural language terms, such as a book's *ISBN*, are considered also as NOLOTs as well. Also, composite attributes– like *Name* in the upper schema of the right side– must be represented as NOLOTs. Therefore, the ORM makes not anymore a rigorous distinction between LOTs and NOLOTs.

The double-headed arrows in Figure 3.10 which are drawn above (or below) the fact types are *uniqueness constraints*. These can be modeled for a single role of the fact type, such as for the role on the right-hand-side of the fact type between *Employee* and *Birthdate* (*Employee* determines *Birthdate*, *Employee*→*Birthdate*), can be modeled for both roles, such as on both roles of the fact type connecting *Employee* and *Ssn*, respectively, which describes *Ssn* as the unique key of *Employee*, or can be modeled on both roles together, such as the double-headed arrow on the roles of the fact type connecting *Employee* and *Skill*, which is describing a *multivalued fact*. The ∧ symbols, for instance on the line between *Employee* and the fact type between *Employee* and *Skill*, describe *all constraints*. That means, that each *Employee* must participate in the set of the fact type between *Employee* and *Skill*, which expresses that each *Employee* has at least one *Skill*. *All constraints* are used to describe the *NOT NULL* property of single-valued attributes as well.

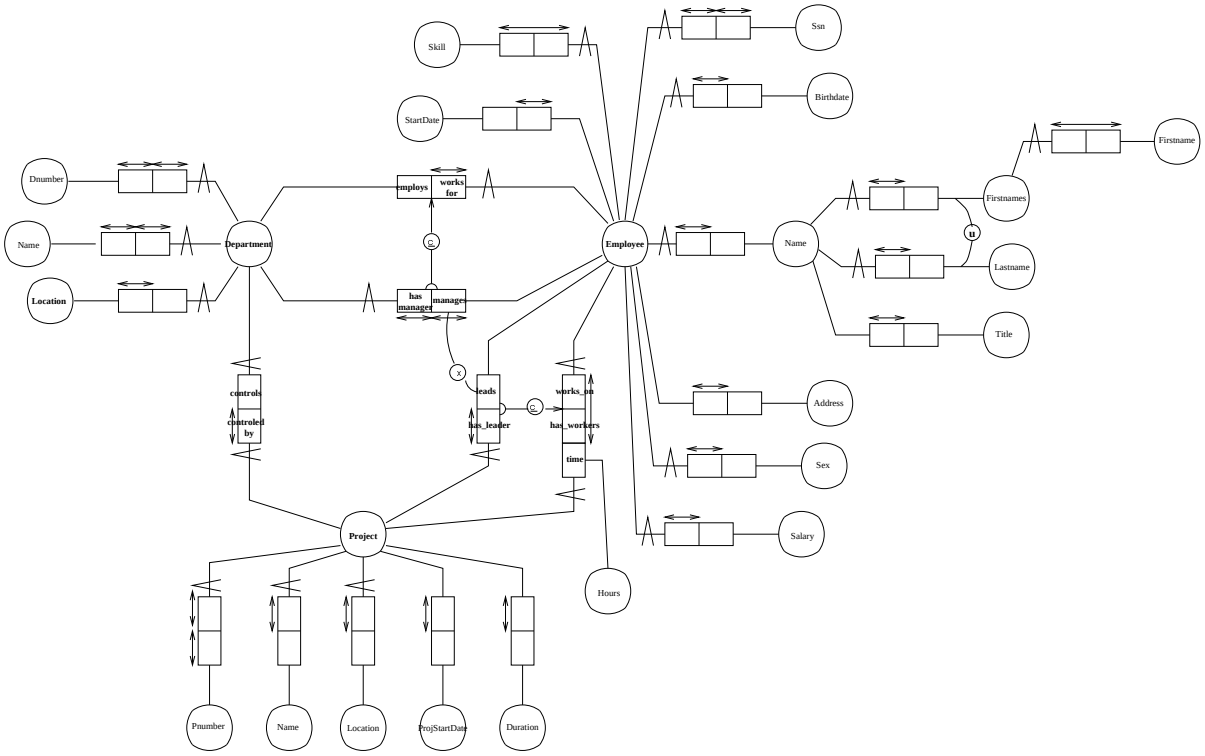


Figure 3.11: ORM Company Schema.

The upper diagram of the right side of Figure 3.10 shows another type of *uniqueness constraint*. The arc with the circle labeled u indicates that the combination of $Firstnames$ and $Lastname$ uniquely determines $Name$ ($\{Firstnames, Lastname\} \rightarrow Name$). If, furthermore, $\{Name\}$ where a key attribute set of $Employee$, then this would describe $\{First\text{- }names, Lastname\}$ as an *alternate key* of $Employee$ (assuming that Ssn is the primary key).

The lower diagram on this side presents further constraint types which are supported by NIAM and ORM. In this diagram we have also annotated the role names on the fact types. We have already noted in Section 3.2.2 and 3.2.3 that the project leader ususally works on that project. This inclusion constraint is represented in the lower diagram by the arrow and the subset ($\subseteq$) symbol, which are between the fact type with roles *leads* and *has_leader* and the fact type with roles *works_on* and *has_workers*. However, such constraints between fact types may be not related to both roles of the fact type (to the combination of the roles, respectively). Constraints may also be related to only one role of the fact types. Suppose, a department manager can never be a project leader the same time. In NIAM and ORM, this is expressed as indicated by the line with the circle labeled $\times$, between the *manages* and *leads* role.

The ORM like the one used in *van Bommel* [Bom95] employs also n-ary ($n > 2$) fact types. Figure 3.11 gives an impression how the entity-relationship schema of Figure 2.1 could look like in the ORM. According the SQL unique index definition in Figure 2.10, we have added here that *Name* is an alternate key of *Department*.

Note, that in the schema in Figure 3.11 *Skill* and *Firstname* are attributes which are connected by a multivalued fact to the *Employee* entity and *Firstnames* attribute, which represents that they are sets (lists). The modeled inclusion contraints ($\subseteq$) and exclusion constraints ($\times$), proposes the database implementation to use triggers for their realization.

3.3.2 Extensions of the Relational Data Model

In this Section we present concepts to specify the invocation of the additional actions which are necessary to restore consistent database states (considering *rule triggering systems,* RTSs), and introduce the nested relational model.

3.3.2.1 Specification and Maintenance of Dynamic Constraints

In the late 80's, the DBMSs invoking additional tasks on database modifications were considered as active DBMSs (ADBMSs). During the last years most relational, object-relational, and object-oriented DBMS vendors have integrated so-called *event-condition-action* (ECA) rule-triggering mechanisms into their systems. That is, as soon as an *event* occurs, such as an insert, delete, or update operation, the *condition* is checked, and, if it is true, the DBMS invokes the *action*. This is called the *rule fires*. It should be clear that the action herself can trigger another action in turn, for reason that another (or maybe, the same) ECA rule fires since its event occurs and condition becomes true, after the first rule fires.

C_1. *The old salary of a person can not be higher than the recent salary of a person, otherwise the persons must distinct.*

C_2. *If the count of passengers of a flight is higher than the capacity of the airplane used for that flight, then reschedule those passengers which are overbooked to other fights and provide information about their rescheduling.*

Table 3.1: Examples of Dynamic Integrity Constraints.

To determine the behavior of the specified database rules is a non-trivial task (refer to [CW90, AHW95]), and there exist some analysis techniques which demonstrate how inconsistent database rule specifications can be detected. E.g., [ST94a, SST94] exhibit on deriving triggering rules automatically from the set of integrity constraints for a given

database schema, and show which problems may occur, especially when different kinds of integrity constraints, such as IDs and EDs, are used together.

On the design level, ECA rules can be specified by integrity constraints like shown in Table 3.1. Generally, these kinds of integrity constraints are called *dynamic*, but we see that the property dynamic is not that the constraint itself is dynamic, and therefore changeable, as one could assume. The constraint (C_1, respectively C_2) itself is rather static, but, the condition which it specifies is dynamic and uses either historical information (on its own value), like done by C_1, or else operates on the database to evaluate whether the condition is true or not, like done by C_2. In either case the constraint may be violated, but the dynamic constraint specifies the *repairing action* to make it valid again.

A problem of first ADBMSs was that too much computing power was consumed whenever the DBMS was waiting for events to occur and conditions to become true. Therefore, ADBMSs and so, all DBMSs of today which include rule-triggering mechanisms, avoid permanent checking if an event of some rule occurs (and the condition is true) by using a *transaction scheduler*. Transaction schedulers control all running transactions of the database (and so, the insert, delete, and update operations). They perform tasks such as calling the transactions according their priority as well. E.g., if a transaction T_1 must be applied before a transaction T_2, then the scheduling T_1 and T_2 after T_1 is given. Is such a requirement not given (T_2 may be applied before T_1), then the scheduler can apply T_2 and T_1 after T_2, although T_1 may have been required firstly by the applications.

Also, some ADBMSs, e.g. HiPAC [DBB+88], use so-called *situation action rules,* to describe the coupling of the transactions. That is, two dependent transactions can be coupled *immediate*, *deferred*, or *separate*, such that they are called by the ADBMS immediately, with refrerence to an according delay, or independently. E.g., the rescheduling of the overbooked passengers in C_2 must be done immediately, but the informing of these passengers according that they are assigned to other flights may be done with a delay (deferred).

The complete execution sequence of such *nested transactions* like described by C_2, may be aborted (rolled back) whenever an error occurs in some child-transaction. In such a case, the tuple which is responsible that a flight becomes overbooked, is not inserted in the database such that the flight also does not become overbooked. (If it is not possible to schedule a passenger who wants to get a flight that is already booked out to another alternative flight, then the passenger can not get any flight.) Such transaction rollbacks can be implemented for the constraint C_1 as well using todays RDBMSs, by implementing an *after-update* trigger which raises an error if not $new.salary \geq old.salary$, for the record before update, *old*, and the record after update, *new*.

3.3.2.2 The Nested Relational Model

The representation concepts for relationships between data, of the hierarchical data model and the network model, respectively, are associations between parent and children records, such that they represent one-to-many mappings that can be assumed as hierarchally organized tables containing many children records for each parent.

In the SQL create table statements for the relational database in Figure 2.10, we have used two such relationship types in the *Employee* relation, where we implemented the *Firstnames* attributes and *Skills* attributes as repeating groups, respectively. However, there is no support in SQL to transfer these flattened lists respectively sets back to their original meaning, such that the application interface must make the user aware how he has to understand and maintain the values of the *Firstnames* and *Skills* columns.

Department	Manager	Employee			
		Name [Firstname]	Lastname	{Skill}	{Project}
Computer Science	Miller	David	Miller	Computers	Documentation Analysis
		Sven	Martin	Computers Philosophy	Development
		Jon	Smith	Engineering Physics	Implementation Analysis
Mathematics	Newman	Mary Ann	Miller	Mathematics	Analysis
		Susan	Smith	Philosophy	Analysis

Table 3.2: Nested Relation combining the Populations of Table 2.9 and Table 2.10.

As a way out of this dilemma, the nested relational model considers the properties of handling record-typed and multivalued data. A nested relation allows each component of a tuple to be either atomic or another nested relation, which may itself be nested several levels deep. According [PBGG89], the nested relational model can be informally described as follows:

1. Attributes, can be either of atomar domains (Section 2.3) or composed; "composed" means that sets of attributes can be composed in turn, as well as that attributes may represent multivalued sets.
2. Relation schemata are sets of the possibly composite or multivalued attributes.
3. And, relations, which are sets of the records described by the relation schemata, and are containing the values of the possibly composite or multivalued attributes.

Figure 3.2 shows the nested relation representing the database population of the relations that were shown in Figure 2.9 and 2.10. One can recognize, that the nested relation maps the following FDs and MDs:

$Department \rightarrow Manager$	(* each Department has one Manager *)
$Department \twoheadrightarrow Employee$	(* each Department employs a set of Employees *)
$Employee \rightarrow Employee.Name$	(* each Employee has one Name *)
$Employee.Name \twoheadrightarrow Employee.Name.Firstname$	(* each Employee's Name consists of Firstnames *)
$Employee.Name \rightarrow Employee.Name.Lastname$	(* each Employee's Name consists of a Lastname *)
$Employee.Name \rightarrow Employee.Name.Title$	(* an Employee's Name may contain a Title *)
$Employee \twoheadrightarrow Skill$	(* each Employee has some Skills *)
$Employee \twoheadrightarrow Project$	(* each Employee works on some Projects *)

such that we have different forms of nesting:

- the MD $Employee.Name \twoheadrightarrow Employee.Name.Firstname$ specifies a list-typed nesting,
- the MDs $Department \twoheadrightarrow Employee$, $Employee \twoheadrightarrow Skill$, and $Employee \twoheadrightarrow Project$ specify set-typed nestings,
- and, the FD $Employee \rightarrow Employee.Name$ represents the unique mapping of each $Employee$ entity to her composite attribute $Name$.

Nested Relations and Normalization. For reason that the relation in Figure 3.2 represents these functional and multivalued dependencies directly, in a partitioned form, that relation is said to be *partition join normalform* (PNF), as defined by [RK87, RKS88]. On the other hand, we see that special values like the *Skills* "Computers" and "Philosophy", or the *Project* "Analysis" are represented more than once in that relation. This is for reason of the hierarchical tree-construction of the PNF relation, and extends if the values ("Computers", "Philosophy", and "Analysis") were not only simply structured, i.e. strings, but were complex objects.

In such cases the maintenance workload for handling the complex objects which are contained in the subrelations (called "buckets") would be high and expensive. Therefore, in [MNE96] a normalform for nested relations, called *nested normalform* (NNF), is presented, which extends PNF and considers these undesired anomalies which are possible in PNF relations.

3.3.2.3 Concepts and Properties of New-Generation DBMSs

Even though most object models– also the ones which were more recently proposed than those which we considered in Section 3.2.1 - 3.2.3 –do not support the representation of record and collection types directly, it may be a little bit wondering that all object-oriented DBMSs (ODBMSs) which were developed since the ending 80's and beginning

90's, and their underlying data models consider these new database types. This may be reasoned by the fact that network DBMSs already considered these requirements, and most ODBMSs owed lot of the tradition of network DBMSs.

Let us survey on some of the new-generation DBMSs. In the following, we give an overview on the *GemStone*, *Ontos*, and O_2 ODBMSs, the *Postgres* object-relational DBMS (ORDBMS)[7], and finally, we consider the *Oracle*8 ORDBMS and *Oracle Web Application Server*.[8]

GemStone has been developed rooted on the Smalltalk object model as a kind of object-oriented database programming language. The global architecture of GemStone is partitioned into (1.) Stone processes, which communicate with the operating system and manage the read/write operations to secondary storage, (2.) Gem processes, which administrate object-identities and offer the application interface, and (3.) client applications. GemStone was developed by a crew headed by David Maier, using *C* as implementation language, and its first commercial version was already released at the end of 1987.

GemStone's schema definition language *OPAL* provides a Smalltalk like structuring of classes, and instance and class variables. Like in Smalltalk, in OPAL user-defined classes are considered as subclasses of *Object*.[9] The predefined classes *Set*, *Bag*, and *Array*, as well as numerous database types, e.g. *DateTime*, *Character*, *Number* and *String*, and the *constraints* definition for the instance variables, give GemStone the DBMS flavor and help to prepare the user-defined classes for database usage. Figure 3.12 shows how the *Employee* class is defined in OPAL, presupposing that the classes *EmplName*, *SetOfSkill*, and *Department* were previously defined by the user.

```
Object subclass 'Employee'
  instVarName: #('ssn' 'birthdate' 'emplname' 'address' 'sex' 'skills' 'dept' 'salary')
  classVars: #()
  constraints: #[ #[#ssn,String],
                  #[#birthdate,DateTime],
                  #[#emplname,EmplName],
                  #[#address,String],
                  #[#sex,Character],
                  #[#skills,SetOfSkill],
                  #[#dept,Department],
                  #[#salary,Number] ].
Set subclass: 'SetOfEmpl'
  constraints: Employee.
```

Figure 3.12: GemStone/OPAL Definition of the Employee Class.

GemStone does not provide generic database operators, like SQL's select[10], insert, delete,

[7] Parts of this overview on new-generation DBMSs are taken from [Heu92] and [Kim95].

[8] The overview on Oracle8 and Oracle Web Application Server is based on [PLSQL98] and [GCS+97].

[9] In Smalltalk, everything is considered as an object, so a type is considered as an object as well.

[10] Do not confuse GemStone's selector method *select* with the SQL-select operator.

and update, but it provides the *constructor* and *selector* methods for objects as provided by Smalltalk. So, if we have created an object of the class *Department* called *deptinst*, then we can retrieve all departments that have employees working for that department which have skill "Mathematics" by the following command:

```
( deptinst select: { :someempl | someempl.dept = deptinst
                                 and someempl.hasSkill: ['Mathematics'] } )
```

In this example, *someempl.dept* delivers the *dept* of the employee pointed to by the introduced instance variable *someempl*, and *someempl.hasSkill* returns true or false according the employee has the skill which is passed as parameter to the *hasSkill* method ('Mathematics'). For the example database population of Table 2.9 and 2.10, or Table 3.2, this expression returns the singleton set which contains the department with name "Mathematics" and manager "Newman".

Properties. All objects which are created are automatically persistent. To hold these data, GemStone uses page and object buffers. It provides physical storage structures for simple objects, like booleans, integers, or object identifiers, as well as for record-typed objects and for collection-typed objects, i.e. for *Bag* with its subclass *Set* and for *SequencableCollection* with its subclasses *Array* and *String*. Objects which are of complex, i.e. non-simple, type can be stored such that its values reside close to each other and in sequence of the attribute order of that class. To invoke this physical storage operation, a method called *cluster* must be called explicitely for that object. GemStone uses *equality indices* which point to instances of simple objects, e.g. integers or strings, and *identity indices* which point to complex objects. Identity indices can only be used to ascertain whether objects are identical or not, but not whether all the values by which they are represented are equal.

GemStone provides two different kinds of transaction management and concurrency control. The first is optimistic concurrency control such that the user works on the physical database that is also used by the other users. The second is pessimistic concurrency control where the system creates a shadow image of the database for each user session, such that updates are only stored on that shadow image, unless the user invokes per *System commitTransaction* the physical transfer of his local image to the global database. Clearly, the *commitTransaction* can only succeed if there is not any conflict or integrity violation. The user can also delete his local image, and start a new session with a fresh image. This is done by means of the *System abortTransaction* command.

Ontos. The client/server architecture of Ontos is similar to that of GemStone, but its database kernel and data model are rooted in the C++ class hierarchy. This way, the basic type constructor of Ontos is given by the class *aggregate*, and *set*, *list*, and *association*

are special subclasses of *aggregate*. These predefined classes have standard methods, for example iterators which are used to traverse the set or list. Ontos considers a class to have one instance (like a module), which has a set of the actual objects as component. By means of a *meta schema* concept, Ontos can represent different relationship types between classes, which can be annotated cardinalities or can be signed as *inverse.* Inverse relationship types can be compared to relational database triggers implementing exclusion constraints, such as setting the $MgrSsn$ attribute to $NULL$ for a *Department* whose manager ($Employee.Ssn$) becomes a *Project* leader. Recall the SQL code presented in Figure 2.10.

As GemStone, Ontos considers user-defined classes as subclasses of *object*; these must have certain properties which include a constructor and some necessary methods. By means of $classify$ commands, classes which are described by the C++ class definition are loaded into the database, and are thereby made persistent. Ontos distinguishes two kinds of object identity. The first is *direct references* which can be compared to the physical address of the object or record, the second is *transparent references* which can be compared to unique key values in relational databases. Accordingly, the objects of the classes are physically either unordered arrays which are accessed by means of *linear hashing*, or so-called *dictionaries* which are implemented by *Btrees*.

The objects are loaded from secondary storage to main memory by the *activate* command, and the content of an object is written back to secondary storage by the *deactivate* command. Ontos provides nested transactions, such that for objects which are *activated* on different levels (separate, in their sets, or including all component objects) different lock levels are set. Then, on occurence of conflicts of different transactions which are accessing the same elements according exceptions and exception handlers can be used to generate and to resolve conflicts, such that transactions are aborted or transactions which recognize incorrect database states or receive exceptions from other transactions can be continued. The start of the transaction can be provided a set of parameters, describing the kind of buffering of the objects or how the possible conflicts shall be resolved.

Unfortunately, Ontos provides no high level data manipulation language, and the transaction definition for these actions is also not described by some declarative language like SQL, such that it is due to the application programmer to use the right C/C++ statements that the objects are read and written appropriately, and no data inconsistences are generated.

O_2. The O_2 system was developed from 1986 as a common project by a crew headed by Francois Bancilhon at INRIA, Paris, and its first commercial version was released in 1991. An important part of O_2 is the *Wisconsin Storage System* (WiSS) [CDKK85], which realizes the secondary storage accesses and page buffers, and is responsible for concurrency

control and recovery as well. Primary storage structures of the WiSS are sequential files, *dense Btrees*, and *linear (extensible) Hashing*.

The *object manager* is that part of O_2 that manages the object identities and records, provides the O_2SQL generic operations, select, insert, delete, and update, and is responsible for the object persistence, *cluster*-structures, and indices. It maps the object representations to the WiSS physical storage representation. The O_2 *schema manager* is resposible for the classes, methods, and the names of the persistent objects. The WiSS, the object manager, and the schema manager build the O_2 engine, which has serveral external database programming language interfaces, such C, C++, O_2C, or O_2SQL, on base of which– in turn –a couple of so-called O_2Tools are implemented.

O_2 Schema Definition and Operations. As basic types, O_2 provides *boolean, character, integer, real, string,* and *bits.* Furthermore, the type constructors *tuple, list, set* (bag), and *unique set* (set) can be used to construct complex types. These types are used to specify the attributes of the classes of an O_2 schema. The type of an attribute can be another, previously defined class as well, which is then looked at as a reference to that other class. Like relational DBMSs, O_2 stores the meta information of the schema, that is the description of the classes, attributes, and methods, by classes again ("data dictionary").

Let us consider a database strucuture which is defined using O_2SQL.[11] Figure 3.13 shows the definition of the classes *Employee*, *works_for*, *ProjLeader*, *ProjWorker*, and *works_on*, assuming that the classes *Department* and *Project* have already been defined.

```
CLASS Employee
  TYPE TUPLE(Ssn: ARRAY(CHARACTER,9) NOT NULL UNIQUE,
             Birthdate: DATE NOT NULL,
             Sex: CHARACTER NOT NULL,
             Name: TUPLE(Firstnames: UNIQUE SET OF STRING
                                                  NOT NULL,
                         Lastname: STRING NOT NULL,
                         Title: STRING),
             Address: STRING,
             Skills: SET OF STRING NOT NULL,
             Salary: REAL NOT NULL);
```

```
CLASS works_for
  TYPE TUPLE(Empl: Employee UNIQUE,
             Dept: Department);

CLASS ProjWorker INHERITS Employee
  TYPE TUPLE(HoursTotalPerWeek: REAL)
  METHOD My_Projects: STRING SET;

CLASS ProjLeader INHERITS ProjectWorker;

CLASS works_on INHERITS ProjectWorker Project
  TYPE TUPLE(Hours: REAL NOT NULL)
  RENAME ProjectWorker.Name TO EmplName
  RENAME Project.Name TO ProjName;
```

Figure 3.13: O_2 classes Employee, works_for, ProjLeader, ProjWorker, and works_on.

The schema definition in Figure 3.13 shows the structural inheritance between the superclass *Employee* and its subclasses *ProjWorker* and *ProjLeader*, respectively, as well as the multiple inheritance of *works_on* according *ProjWorker* and *Project*. Because of the name conflict according the *Name* attribute which is part of the *ProjWorker*

[11] O_2SQL did also serve as basis for the ODMG [Car94] specification of the object database definition and query language OQL.

(*Employee*) class as well as of the *Project* class, this conflict is resolved by renaming these attributes in the subclass *works_on*.[12]

There exist a couple of operators for each of the types and type constructors, such as unique addition of a new element to a set, or iterators for working with those elements of a set (or list, or bag) for which a predicate holds, e.g. *for <elem> in <set> where <predicate>*. In the example of Figure 3.13, the method *My_Projects* of the *ProjWorker* class is only described by its signature. Its implementation must be given in C, C++, or O_2C. The following code shows how it is implemented in O_2C, which includes O_2SQL:

```
METHOD BODY My_Projects: SET(STRING) IN CLASS ProjWorker
  { RETURN
      SELECT p->ProjName
        FROM p IN works_on
       WHERE p->Ssn = self->Ssn }
```

Here, *self* denotes the *ProjWorker* himself, who inherites the attribute *Ssn* from the *Employee* class; the *works_on* class inherits the attributes *Project.Name* (as *ProjName*) and *ProjWorker.Ssn*. Attributes of an O_2 schema can also be marked *READ*, such as, for instance, *Project.Pnumber* which should be read only, and *PRIVATE* or *PUBLIC*.

To provide efficient access paths to the objects, O_2 also supports– as mentioned above –indices, and *cluster trees* which assign a collection of objects to a special part of the physical storage. E.g. the command

```
CLUSTER TREE FOR CLASS works_on ASC Pnumber
```

causes the O_2 DBMS to place the *works_on* records physically close to the other *works_on* records of the same *Project* (presupposing that *Projects* are identified by *Pnumber*).

Postgres. Postgres was developed by a crew headed by Michael Stonebreaker since 1986, as the successor of the Ingres DBMS. Some concepts developed in the Postgres project had also influence on the commercial Ingres releases 6.3 and 6.4, and many new features, like collection-typed and user-definable attributes, and encapsulation concepts, which are used together to form ADTs, were used in the OpenIngres version which is distributed by Computer Associates (CA, see [ASK94]), as the latest Ingres versions were. Also, since the mid of the 90's commercial versions based of the root Postgres technology were available with Illustra, that has meanwhile been bought by Informix. CA's current object-oriented DBMS is Jasmine, which could be looked at as a successor of OpenIngres and was developed by a group headed by Alan Gupta, a well-known developer of RDBMS technology.[13]

[12] Although here both inherited *Name* attributes were renamed, it would suffice to rename only one of the *Name* attributes to keep the schema consistent and avoid the name conflict.

[13] That Jasmine is a successor of OpenIngres is my impression, since they use terms in the Jasmine publications which I did already see in the OpenIngres publications, but nowhere else.

The type system of Postgres is based on the data types traditionally used by RDBMSs, and furthermore, the [n] and [] type construtors, which are used for fixed-length and variable-length arrays, and the object identifier attribute type (OID), whose values are used to reference tuples of other relations. The OID values are system maintained and can only be read by the application, although they may be used as foreign keys. Postgres distinguishes base relations from derived relations, such that derived relations are defined in a view-like manner and can be used as attribute values in other relations. Postgres allows single and multiple inheritance, such that the latter excludes conflicts by forbidding inconsistent hierarchies. The hierarchy of the database structures, however, is a type hierarchy which supports tuple types only.

```
LOAD '/home/steeg/WWW/docs/T_EName.so';
CREATE FUNCTION EName_in(opaque) RETURNS EName
 AS '/home/steeg/WWW/docs/T_EName.so' LANGUAGE 'c';
CREATE FUNCTION EName_out(opaque) RETURNS opaque
 AS '/home/steeg/WWW/docs/T_EName.so' LANGUAGE 'c';
CREATE TYPE EName (INTERNALLENGTH=34,INPUT=EName_in,OUTPUT=EName_out);
-- SQL does not distinguish uppercase/lowercase: EName_in -> ename_in (in C)
CREATE TABLE Employee (
  -- Ssn DECIMAL(9) NOT NULL,
  -- Postgres does not support DECIMAL as type of key attributes
  Ssn CHAR(9) NOT NULL,
  Role CHAR(1) NOT NULL CHECK (Role IN ('m','l','w','s','a')),
  Birthdate DATE NOT NULL,
  Sex CHAR(1) NOT NULL CHECK (Sex IN ('m','f')),
  Name EName,
  Address VARCHAR(40) NOT NULL,
  Skills VARCHAR(20)[] NOT NULL,
  DeptName VARCHAR(20) NOT NULL,
  Salary FLOAT,
  PRIMARY KEY (Ssn)
);
-- Ensuring that DeptName is really in Department:
--  we have defined a procedure check_foreign_key()
CREATE TRIGGER DeptNameInDept AFTER INSERT OR UPDATE ON Employee FOR EACH ROW
 EXECUTE PROCEDURE check_foreign_key(OID,'Employee','DeptName','Department','Name');
CREATE VIEW ProjLeader AS
 SELECT Ssn, Birthdate, Sex, Name, Address, Skills, DeptName, Salary
   FROM Employee WHERE Role = 'l';
-- Defining the Insertion Rule for ProjLeader
CREATE RULE pl_ins AS ON INSERT TO ProjLeader
 DO INSTEAD INSERT INTO Employee VALUES (NEW.Ssn,
    'l', -- marks that the Employee is a ProjLeader
    NEW.Birthdate,NEW.Sex,NEW.Name,NEW.Address,NEW.Skills,NEW.DeptName,NEW.Salary);
```

Figure 3.14: PostgreSQL Creation of Tables, Views, Triggers, and Rules.

Integrity Maintenance and Operations. In contrast to most other object-oriented DBMSs, Postgres includes trigger definitions for integrity control and invocation of additional actions as well. Beyond the usage of triggers used in most RDBMSs of today which are specified for inserts, deletes, and updates on single relations, in Postgres triggers as well as rules can also be defined for views. The native query language of Postgres, POSTQUEL, which is the successor of the QUEL that was used by Ingres, supports recursive queries as well. Meanwhile, POSTQUEL has been substituted by PostgreSQL, but PostgreSQL

took over many of its powerful features. So, the powerful *Rule-System* which provides– beside the triggers –the creation and maintenance of any kind of updatable view, such that views can be used to perform the correct actions on the base tables.

Example. The definition of the *Employee* relation, the *ProjLeader* view, and the *Rule* which rewrites inserts on the view *ProjLeader* to the *Employee* base table, is shown in Figure 3.14.[14]

In PostgreSQL, it is also simply possible to specify subtyping of relations, by means of an *INHERITS* clauses that is appended to the *CREATE TABLE* statement. However, the construction of composite types, like the *EName* which is used in Figuer 3.14, or of array types that do not have collection elements based an simple SQL types, is a little bit troublesome. The usage of these attribute types is not as simple as, for example, demonstrated by the *Employee* class definition in Figure 3.13. But it must be noted that Postgres is not a commercial DBMS, but a prototype.

Oracle8 and Oracle Web Application Server. Like GemStone, Ontos, and O_2, Oracle is a commercial DBMS. Oracle8 is an object-relational DBMSs which is extending the features of the latest Oracle7 release (Oracle 7.3.4). The features which were already contained in, or which can be looked at as orthogonal to features implementable in the programming language interface (*PL/SQL*) of the previous release, include:

1. Packaging. Packages allow to bundle logically related types, variables, cursors, and procedures, which are used as shared resources in application development. They consist of (1.) a *specification* interface to the application (signature), and (2.) a *body* which implements the specification. Packages are built by means of the CREATE PACKAGE command.
2. Database procedures and functions. An Oracle database procedure is a PL/SQL block that implements a subprogram which can be called from other parts of the database. Figure 3.15 shows how a procedure *make_manager* could be implemented.

   Like in PostgreSQL, in Oracle8 a database function provides an interface to a database module that is implemented in a second- or third-generation-language run-time library. Functions are registered by means of the CREATE FUNCTION command. PL/SQL procedure as well as function implementations may contain code for handling and/or raising exceptions.
3. Redefining database inserts, deletes, or updates. Like shown by the CREATE RULE pl_ins AS ... statement in Figure 3.14, the user of Oracle8 has the possibility to redefine inserts, deletes, or updates as well, such that other commands are issued instead of them. This is done by means of CREATE TRIGGER commands which

[14] For some details on the implementation of this example, see Appendix C.3.

```
PROCEDURE make_manager (EmpSsn DECIMAL(9)) IS
  DeptNumber DECIMAL(6);
  OldSsn DECIMAL(9) := NULL;
  ...
BEGIN
  SELECT d.Dnumber,d.MgrSsn INTO DeptNumber,OldSsn
    FROM Department d,Employee e WHERE e.DeptName = d.Name;
  IF OldSsn IS NOT NULL THEN
    IF OldSsn <> EmpSsn THEN
      ...
    END IF;
    ...
  END IF;
  UPDATE Department SET MgrSsn = EmpSsn, MgrStartDate = TODAY WHERE Dnumber = DeptNumber;
  UPDATE Employee SET Role = 'm' WHERE Ssn = EmpSsn;
EXCEPTION
  WHEN %NOTFOUND THEN
    raise_application_error(-17234,'Internal error make_manager (MgrSsn '
                                    || TO_CHAR(EmpSsn) || ')');
END;
```

Figure 3.15: Oracle Procedure Definition of *make_manager*.

contain an INSTEAD OF clause. By this, it is for instance possible to define *general view inserts* which are performing the proper operations on the base tables of that view.

The *data abstraction* (ADT) features which are absolutely new in Oracle8, include:

1. Collection types. Oracle8 supports the collection types TABLE and VARRAY (variable-size arrays) which may be used in the attribute definitions of the create table commands.
2. Record types. Record types can be defined in PL/SQL as well, can have attributes of collection types, and may be nested in turn– by means of including attributes of other record types. This is unlike in Postgres, where record types (like the *EName* type, Figure 3.14) must be realized using dynamic link libraries that are always implemented in an external language.
3. Object types. Like in Postgres, object types in Oracle8 are defined and made persistent by means of create table commands. However, the create table commands in Oracle8 may contain MEMBER PROCEDURE and MEMBER FUNCTION clauses as well, which are used to attach instance methods to the class and give the data definition language a proper object-oriented flavor.

Oracle Web Application Server. The Web Server *cardridges* distributed by Oracle provide interfaces which simply allow to construct Web pages (HTML and/or JavaApplets) that interact with the Oracle database. Here, PL/SQL data manipulation commands which do not include the creation of new and drop of existing database objects, like tables, indices, triggers, procedures, or functions, are sent to the Oracle database server. Then, the select, insert, delete, or update is performed on the database, such that subsequently

the results of the select query invoked by the user are displayed by a (new) form in the browser, or the user is notified about the success of his action, that triggered an insert, delete, or update on the remote machine.

3.3.2.4 The "D" Database Model

The "D" database model was proposed Darwen and Date [DD95] to give firm direction to future database management development. The authors believe that the new generation of database management systems must be firmly rooted in the relational data model and consider object-oriented features of databases as well, but they restrict that *attemps to move forward, if it is to stand the test of time, must reject SQL unequivocally.*

"D" can be considered as a framework defining schema implementation aspects, using future DBMSs. So, "D" is not a model for designing a database, that has a graphical notation and defines the constructs which are to be used, but rather describes an abstract form of a data model, by giving *prescriptions*, *proscriptions*, and *very strong suggestions* for that. These are taken from the relational data model and the different object-oriented approaches to database design:

The Prescriptions from the Relational Model include that

1. domains are described by sets of values,
2. values can be scalars (which are described by single attributes like in the relational model, i.e. they need not to be embodied by an object),
3. tuples are constructed from values, such that the nth value of the tuple is described by the according nth domain and the n domains (and values) of the tuple need not necessarily be distinct (this way, a named attribute does not describe a domain, but rather a type, like boolean, integer, string of length 20, ...),
4. operators like equality ($=$,$\neq$) and those which embody or evaluate to truth values (like $true$, $false$, not, and, or, in, $if\text{-}then\text{-}else$, ...) as well as traditional infix operators for numbers ($+$, $-$, $*$, $/$, $<$, $\leq$, ...) must be predefined,
5. relations are sets of tuples, such that the relation is described by a *heading* which is the domain of each tuple,
6. relation variables $relvars$ are variables described by a domain that is the heading of relation (a $relvar$ is called *base* if its heading is given by relation, or *derived* if its heading is defined for some evaluation procedure on the database), and, a set of integrity constraints, each of which may be related to one or more than one $relvar$,
7. a database variable $dbvar$ is a set of $relvars$,
8. transactions interact with exactly one $dbvar$, and distinct transactions can interact with distinct $dbvars$, such that distinct $dbvars$ are not necessarily

disjoint, and, transactions can dynamically change their *dbvars* by adding or removing *relvars*,

9. operators for *create* and *destroy* of domains, variables (*relvars*, or *dbvars*) and integrity constraints must be provided, such that the *create* operator associates a candidate key to each *base relvar*,
10. relational algebra operators (e.g. π, σ, $\bowtie$) must be supported, and it must be permitted to assign the evaluation of the operator's invocation to tuple variables or *relvars*,
11. operators to create and destroy named *functions* by means of specified relational expression shall be support, and the invocation of a *function* defined this way must be permitted within relational expressions as well,
12. and, every *dbvar* shall include a set of *relvars* describing the structure of the *dbvar*; it means: a data dictionary or *catalog*.

The Prescriptions from the Object-oriented Model include that

1. invocations of functions, database procedures, etc. must be checked at *compile-time*,
2. *single* and *multiple inheritance* shall be supported,
3. "D" shall be *computational complete*, that includes that each result of a database evaluation (which is stored in a *relvar*) must be usable as subexpression in other evaluations,
4. and, "D" shall support *nested transactions*, e.g. it shall support that a transaction T_1 starts another transaction T_2 such that T_1 and T_2 interact with the same *dbvar*, but may be executed asynchronously, and the abort (*rollback*) of T_1 respectively T_2 causes that the other transaction does not generate a new database state, although the other transaction may have commited before.

The Proscriptions from the Relational Model describe that no construct of "D" depends on the ordering of the attributes in a relation or on the ordering of the tuples in a relation *(set-semantics !)*, and whenever two tuples t_1 and t_2 of a relation are distinct, then there must be at least one attribute which carries a different value for t_1 and t_2. "D" must support *nullalble* attributes (attributes which can carry a $NULL$), and no *candicate key* of a relation shall include a *nullable* attribute. There shall be no construct which relates to the "physical" or "storage" or "internal" representation of a tuple.[15] There shall also be no *one-tuple-at-a-time* operations on relations, like today's insert operation. However, if the insert operation wants to add only one tuple to a relation, then this could be performed by assigning the union of itself and a set which does contain only this one

[15] Like the hierarchical and the network data model do, and, as we will see, some object-oriented DBMSs include.

tuple to the relation. One-tuple-at-a-time delete, update, and retrieval operations shall be categorically forbidden, and "D" shall also not include specific support for "composite domains" or "composite columns"– since such functionality is already included by the domain support that is desribed by the three first prescriptions from the relational model.

The Proscriptions from the Object-oriented Model require that a *relvar* should never be considered as a domain and no value (scalar or any other kind) shall possess any kind of ID that is somehow different from a value of the real world.[16] "D" shall also not include concepts like "protected" or "friends" instance variables which are included in object-oriented programming languages, since these properties could be managed by the system's authorization mechanism.

The very strong suggestions from the Relational Model include that is shall be possible to specify one or more candidate keys for a *relvar* such that one is chosen as the *primary key* for the relation, and it should be possible to generate key values by the system, when necessary. "D" shall support $referential\ constraints\ (foreign\text{-}keys)$ and referential actions, such as *cascading deletes*. Furthermore, the ability is desirable (although not completely feasible) to $infer$ candidate keys for every relation R in D, such that candidate keys of R become candidate keys of R' when R is assigned to R', and information about these candidate keys of R should be made available to the user of D. "D" should provide (1.) some convenient quota queries, like "find the three youngest employees", such that those queries are not bundled to an ordered list, (2.) convenient means of expressing the *generalized transitive closure* of a graph relation (e.g. a $Person$ relation which has a foreign-key attribute for the $Person$'s father and mother), including the features of generalized concatenation and aggregation, and (3.) parameters of relation-valued functions to represent tuples and scalars. A mechanism for dealing with "missing information" should be provided as well. **SQL** should be implementable in "D". This shall not stress the fact that "D" becomes a superset of SQL, nor that "D" should be an extension of SQL, but rather that a "converter" translates SQL-code into "D"-code and the evaluations in "D" are transferred back into the SQL format. This way, it may be desirable to have cross-compilers which translate "old application code" with statements in SQL or *Embedded SQL* into new application code with statements in "D".

The very strong suggestions from the Object-oriented Model describe that some form of *type inheritance* shall be supported which was already described by the second prescription from the object-oriented model above, but, "D" should not include a priciple of implicite type conversion (it should be typed), and also not, that functions have a special *receiver parameter*.[17] Constructors for "collection" types, *list*, *array*, and *set*, shall be supported by "D", such that if C is a collection type other than *relation*,

[16] No artifical keys.

[17] Refer to the according Section in the description of the Coad/Yourdon Model [3.2.3].

then conversion functions shall be supported for transforming C into a relation type (transforming the values of C into a relation), and transforming relation types into C (transforming a relation, which has the necessary properties, into values of C). And, "D" should be based on *single-level storage*. That is, there should be no difference whether a piece of data resides in main memory, cache, secondary storage, etc.

3.3.3 The Data Model used in the RADD Approach

The data model used in the RADD workbench is based on the HERM extended entity-relationship model. In this Section we will present firstly the HERM model, and show then how it is used in the RADD workbench and what are the differences between the graphical RADD data model and the HERM.

3.3.3.1 The HERM Data Model

The *Higher-order Entity-Relationship Model* (HERM) [Tha89, Tha97] is based concepts of extended entity-relationship and relational data modeling concepts. It models database structures such that advantageous data representation concepts are considered, like that which are included by the nested relational model presented in Section 3.3.2.2. In the HERM, integrity constraints can be specified as expressive as the those of the ORM presented in Section 3.3.1 as well.

In HERM, the following are considered as structures:

1. Attributes, which are identified by a name that is unique according the entity or relationship type, and are either of usual flat type *(boolean, string, numeric, int, float, date),* composite *(record-typed)*, or nested *(list, set, bag).* Attributes are nullable or not *(WITH NULL,NOT NULL).*
2. Entity types, which consist of a name, a non-empty set of attributes S, called schema, and a non-empty set of key attributes K, such that the key attributes are a subset of the schema, $K \subseteq S$.
3. Relationship types, which consist of a name, a set of attributes s, a set of referenced structures r (called *parent structures*)– which are *entity, relationship, and cluster types,* respectively –, and a non-empty set of key attributes k. References can be annotated role names, integrity constraints, a "K"-item which models that the reference transmits the key from the parent structure to the child structure, and can be nullable or not *(WITH NULL,NOT NULL).* A nullable reference can not transmit the key from the parent structure to the child structure. The key of a relationship, k, is a subset of the union of s and the key attributes of those parent structures in r from which the relationship type inherits the key (indicated by the "K"-references). In

the graphical representation of the HERM, references are indicated by an arrow from the child structure (subtype) to the parent structure (supertype). A relationship type is said *order-1* if all parent structures are entity types (entity types can be looked at as order-0 relationship types), otherwise the order of that relationship type must be greater than the order of all parent structures, respectively.

4. Cluster types, which are union types and comprise a set of references to entity or relationship types, which has at least two elements. In the HERM, cluster types are considered as parts of relationship types, such that a cluster type is referenced by a relationship type. This way, the relationship type that has a reference to the cluster type is used to map the attributes of that union type. The semantics of a cluster type $C = \{R_1, ..., R_n\}$ is that each element of the occurence set of C is either R_1 or ... or R_n, such that the occurence sets of the R_i are disjoint, respectively *(distinct union)*. This way, a cluster type may not be defined for some subtypes of a common supertype because these are not necessarily distinct. The *order* of cluster types is defined analogously to that of relationship types.

Hierarchical database schemata. By the *order* property, a relationship or cluster type can not have a reference to itself, nor can it transitively reference itself, such that the structures of a HERM schema are always hierarchically ordered.

Figure 3.16 shows the HERM representation of the Company Schema. Here, the nodes labeled $DeptManager$, $ProjLeader$, $ProjWorker$, $Secretary$, and $Assistant$ represent *weak entity types*; that is, they model objects which have no own identification, but inherit it from other objects, namly from the $Employee$ entities. However, for reason of this property they are looked at rather as *relationship types* by the HERM, and normally, they are characterized only by a relationship type that has only one reference, called *unary relationship type*.

To illustrate the differences between the modeling concepts of the traditional ERM and most extended ERMs, and the HERM, let us take a look at Figure 3.17. In the left schema of Figure 3.17, the $Student$ subtype is connected to the supertype $Person$ by an *is-a* relationship type. In the right HERM schema, $Student$ is a relationship type that has a reference to $Person$. The left schema forces that the attribute $StudNr$ is chosen as primary key of $Student$, whereas the right schema leaves it open whether to choose $PersNr$ or $StudNr$ as primary key of $Student$. Besides that, the right HERM schema is more compact, and, in our eyes, this is also a more obvious and direct representation of that subtyping relationship.

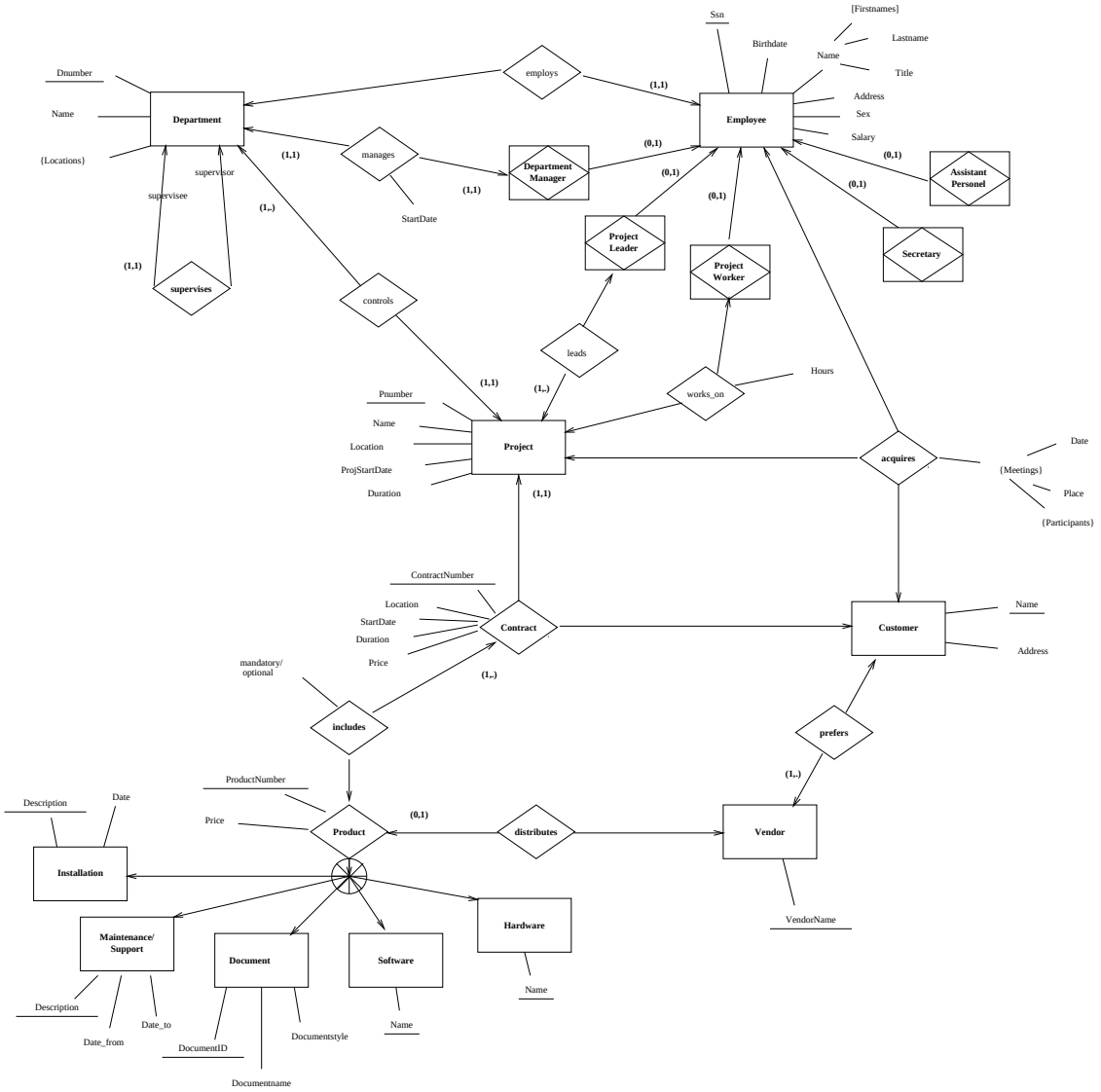


Figure 3.16: HERM Representation of the Company Schema.

The following surveys on the integrity constraint types that are included by the HERM:

1. Cardinality constraints (CCs) are defined as participation constraints, according the cardinality constraints in Section 2.3.4.1.
2. Functional, inclusion, and exclusion dependencies (FDs,IDs,EDs) are specified in the usual way which is known from the relational data model, and are as defined in Section 2.3.1 to 2.3.4.
3. Key dependencies (KDs), as a special case of FDs, are in the graphical HERM design represented by the combination of the underlined attributes (key attributes) and key attributes which are inherited by means of the "K"-marked edges (references), as mentioned in the description of the structures.

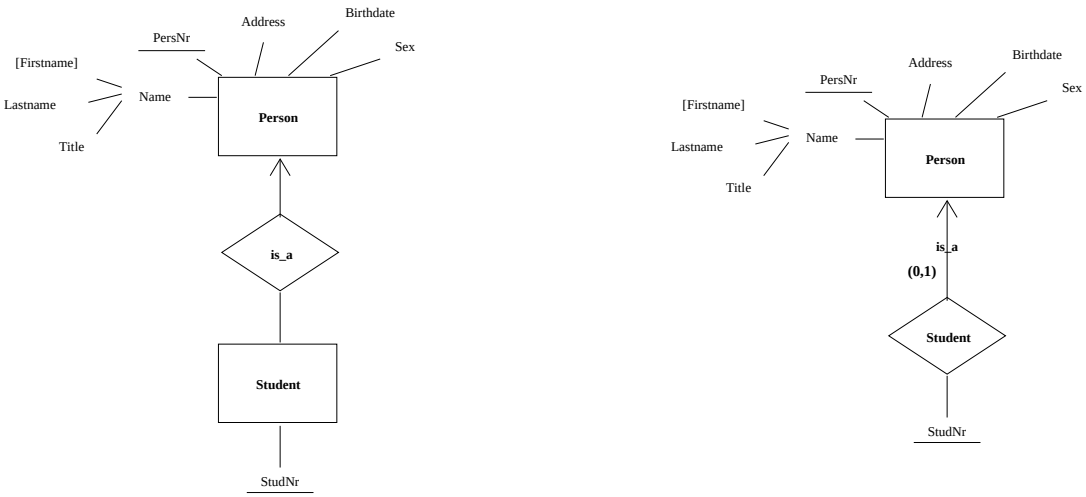


Figure 3.17: ERM- and HERM-Representations of Subtyping (is-a) Relationships.

4. Afunctional dependencies (ADs) specify the opposite of functional dependencies in sense that there are only relations between the instances of the left-hand-side attributes and right-hand-side attributes that violate the according FD. This is like the definition of ADs in Section 2.3.4.

   Another definition of ADs is given in [Alb94]. It considers an AD $X \not\rightarrow Y$ as a constraint such that least two tuples t_1 and t_2 exist, where $t_1[X]{=}t_2[X]$ and $t_1[Y]{\neq}t_2[Y]$. The definition of [Alb94] does not require that all distinct tuples must be different on the right-hand-side (Y), whenever they match on the left-hand-side (X).

5. Path dependencies state a more general kind of the above dependency types (CCs, FDs,IDs,EDs,KDs,ADs). The generalization is that the objects of the dependency (left-hand-side and/or right-hand-side) do not only state a single structure, like entity, relationship, or cluster type, but can be a combination of several structures whose instance sets are considered to be joined. An example of a path constraint is a database consisting of

   - entity types bus-driver, bus, bus-type, driving-licence,
   - and relationship types drives, has-type, has-licence,

   such that we can specify the following path constraint:

   > The bus-driver who drives the bus must have a driving-licence for the type (bus-type) of that bus.

   We will show in Section 6.3 how path constraints are represented by terms in the RADD/raddstar data model. For more information on path constraints the interested reader may also refer to [Tha91] or [Tha97].

Figure 3.18 shows how integrity constraints can be graphically represented using HERM.

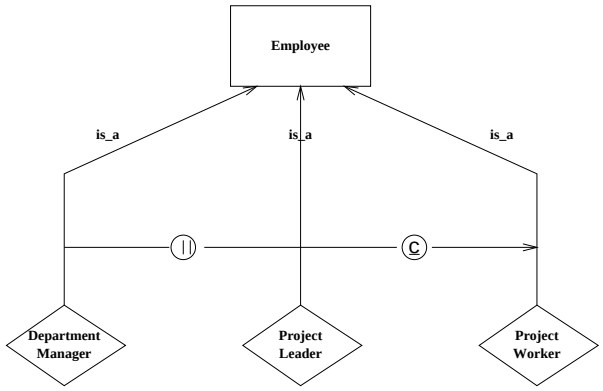


Figure 3.18: Graphical Modeling of Integrity Constraints using the HERM.

The schema in Figure 3.18 omits "K"-labels on the "is_a"-edges from the subtypes *Department Manager*, *Project Leader*, and *Project Worker* to the supertype *Employee*. However, the inheritance of the key of *Employee* to its subtypes is given implicitely by the "is_a" relationship types.

Database Operations. In contrast to the (new) conceptual data models which we have presented above, HERM considers operations as well. In the HERM, a conceptual schema is specified by a triple $S = (Struc, \Sigma, Ops)$ modeling the structures, semantics, and operations of the database, respectively. Generally, it can be assumed that the operations are generated on base of the structures and integrity constraints such that $Ops = GenericOps(Struc, \Sigma)$.

E.g., for a relational database schema which is specified by SQL create table statements (like the ones in Figure 2.10) the operations select, insert, delete, and update are directly generated on basis of the structure of the database relations. This way, whenever a relation $R_1 \in Struc$ references a relation $R_2 \in Struc$, then for a tuple of $t_1 \in R_1$ which references a tuple $t_2 \in R_2$ (by means of a foreign-key), t_2 must be inserted before t_1, such that we can say that $insert_{R_1}(t_1)$ specializes to a *transaction*:

$$insert_{R_1}(t_1) ::= insert_{R_2}(t_2);\ insert_{R_1}(t_1)$$

Recall the SQL code examples and their descriptions which we have presented in Section 2.3.5.1. In the create table statements of Figure 2.10, the *Employee* relation had the column *DeptName* implementing an "application-maintained" foreign-key to the *Department* relation, and the trigger *EmpTriggerDept* was used to ensure that each *DeptName* value of the *Employee* relation is *Department.Name*. This way, a *Department*

must be inserted before an *Employee* is inserted, who is working for that *Department*. This way, we could say that

$$insert_{Employee} \quad \text{specializes to} \quad insert_{Department}; insert_{Employee}.$$

The generation of such specializations of single transactions (insert, delete, update) to "complete operations" were already considered by the *GCS* approach of [ST92] and [ST94b]. This has been implemented in the RADD/raddstar system by means of *transaction extensions*, which we will present in Section 5.3.2.

3.3.3.2 The Data Model of the RADD Workbench

The data model used in the RADD workbench has small differences to the HERM, reasoned by the concepts of the graphical editor that was used as basis for RADD. In the RADD schema editor, the structures of the schema are represented by nodes, and the references of the relationship and cluster types, as well as the attribute connections (in the *attribute tree view*) of the entity and relationship types are represented by edges. This way, the RADD schema editor can not graphically represent edges between edges such that the ID and the ED of Figure 3.18 are represented as shown by Figure 3.19.

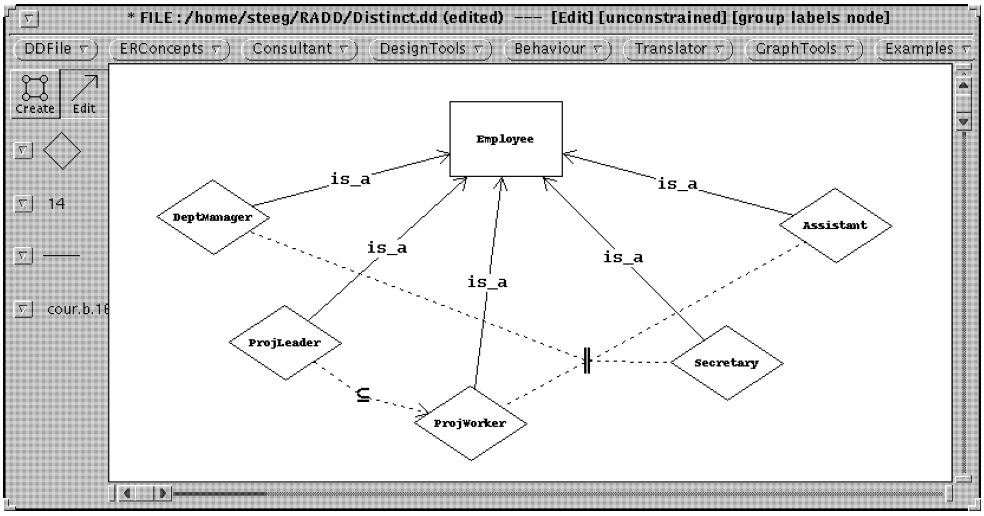


Figure 3.19: RADD Representation of Integrity Constraints.

In Figure 3.19, the subset node ($\subseteq$) denotes the ID

$$ProjLeader{\rightarrow}Employee \subseteq ProjWorker{\rightarrow}Employee,$$

and, the distinct node ($\|$) models the ED

$$DeptManager{\rightarrow}Employee \| ProjWorker{\rightarrow}Employee \| Secretary{\rightarrow}Employee \| Assistant{\rightarrow}Employee.$$

As mentioned in Section 3.3.3.1, ADs are in [Alb94] defined differently from that definition given in Section 2.3.4.3 and used in [Tha97]. The definition of [Alb94] is the way, ADs ase considered by RADD. (Refer to [AAB+95, AAS97a, AAS97b].)

RADD Database Modeling Examples. In Figure 3.20, 3.21, and 3.22 we show screen shots illustrating the design of the Company Schema using RADD.

- Figure 3.20 shows the complete Company Schema, additionally including a *Contract* part, which shows that a project is "acquired" from a "customer" by an employee of the company. Furthermore, it shows that a contract "includes" "products" which can be "installation" and "maintenance" (or support) tasks, technical "documents", "software", and "hardware". If these products are delivered by a third party– like hardware and software, which are bought from another company – then the customer may "prefer" a "vendor" who can "deliver" that product.

- Figure 3.21 shows the *attribute view* of the *Employee* entity type. In the lower left subframe, we see the *Employee* entity type and the structuring and datatypes of its attributes. We recognize that the database designer modeled $Employee.Name$ as record-typed attribute, where the list-typed $Firstnames$ attribute has again a $Firstname$ sub-attribute. The upper right subframe shows the index of all *Employee* attributes, and the lower right subframe shows the attribute editor where the database designer maintains the Ssn attribute.

- Figure 3.22 gives another view, which shows the *acquires* relationship type and its attributes, in the lower left subframe.

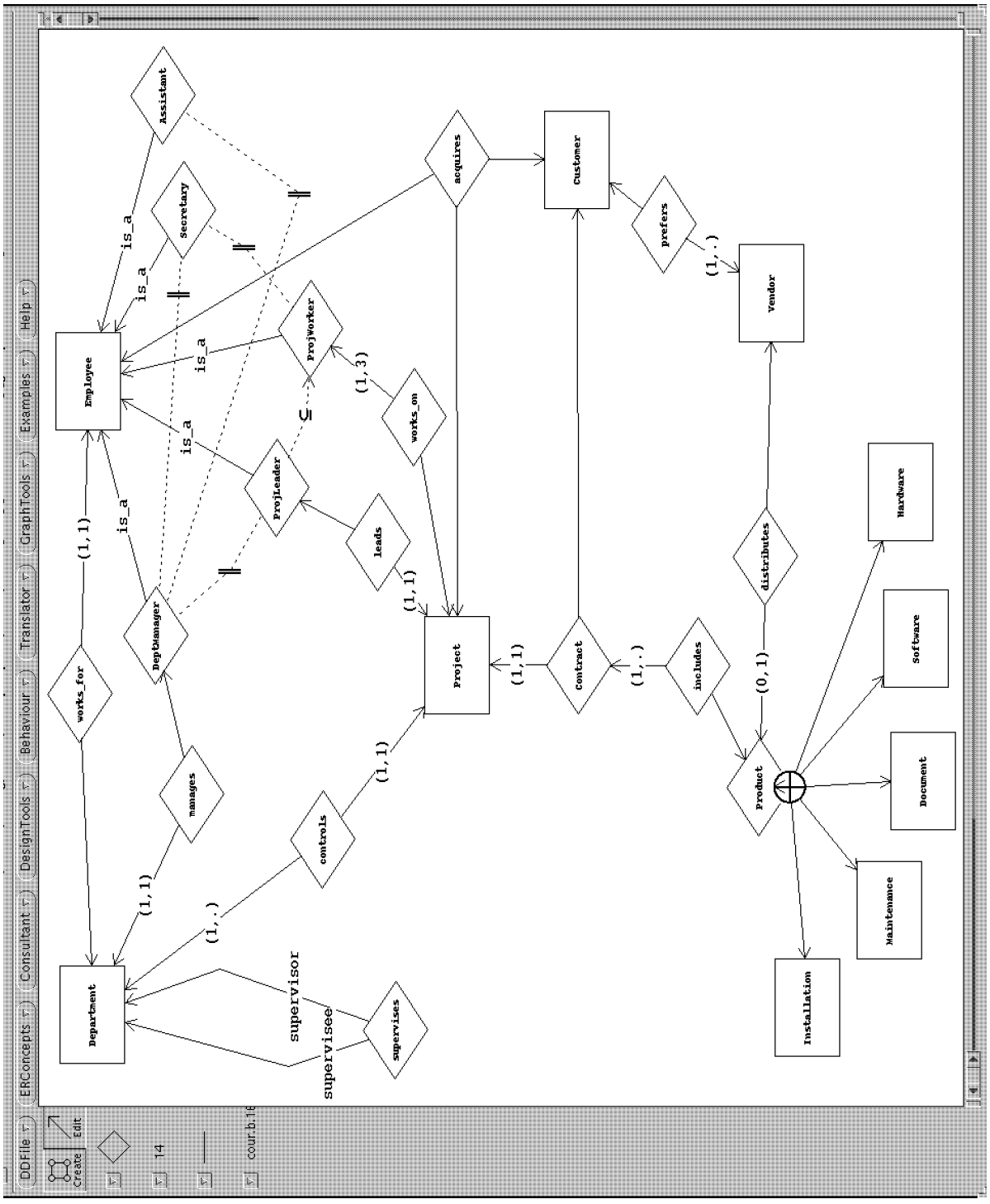


Figure 3.20: RADD Representation of the Company Schema.

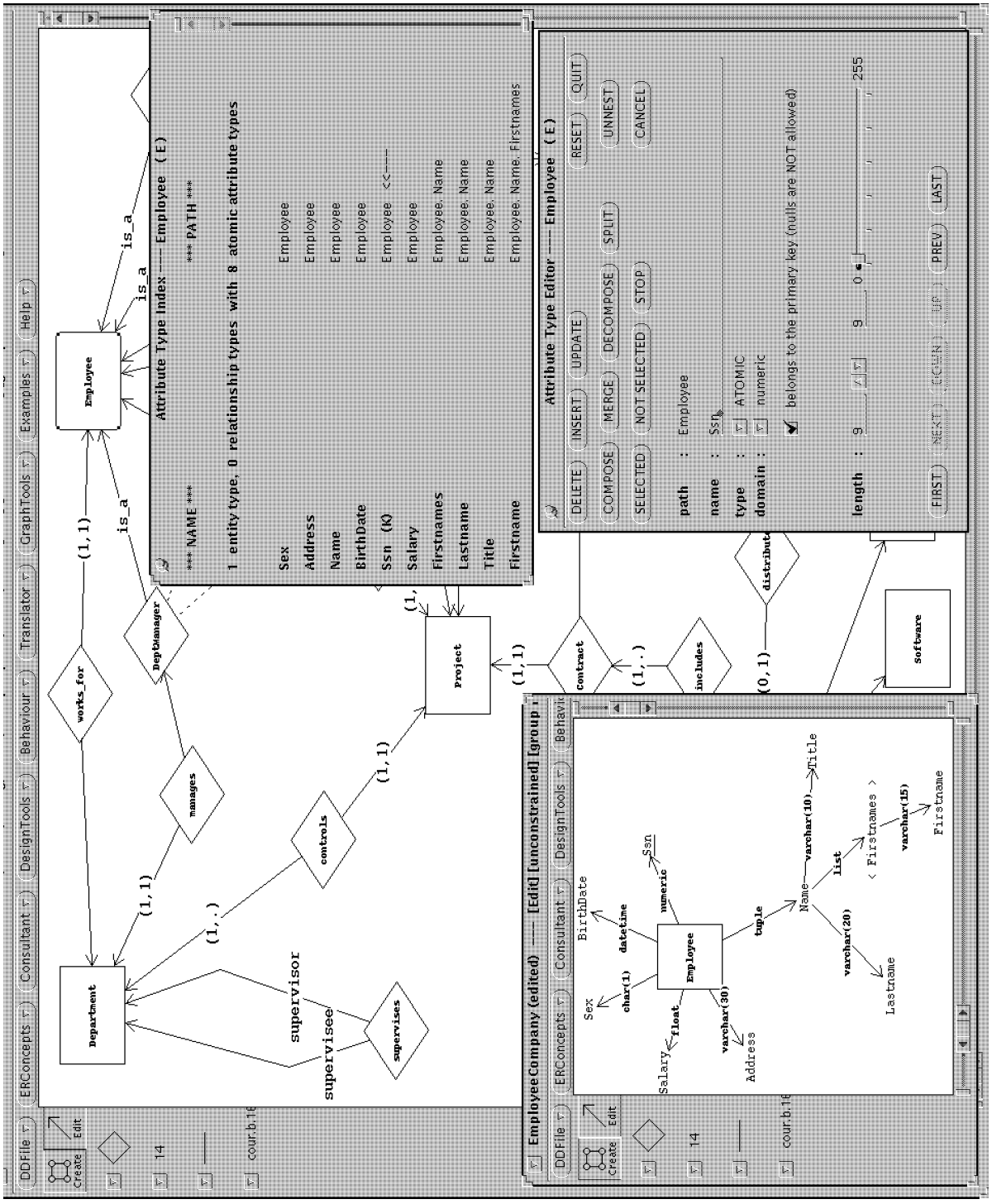


Figure 3.21: RADD Representation of the Employee Type, including the Attribute View on the Employee Type (lower left corner), and the Attribute Editor.

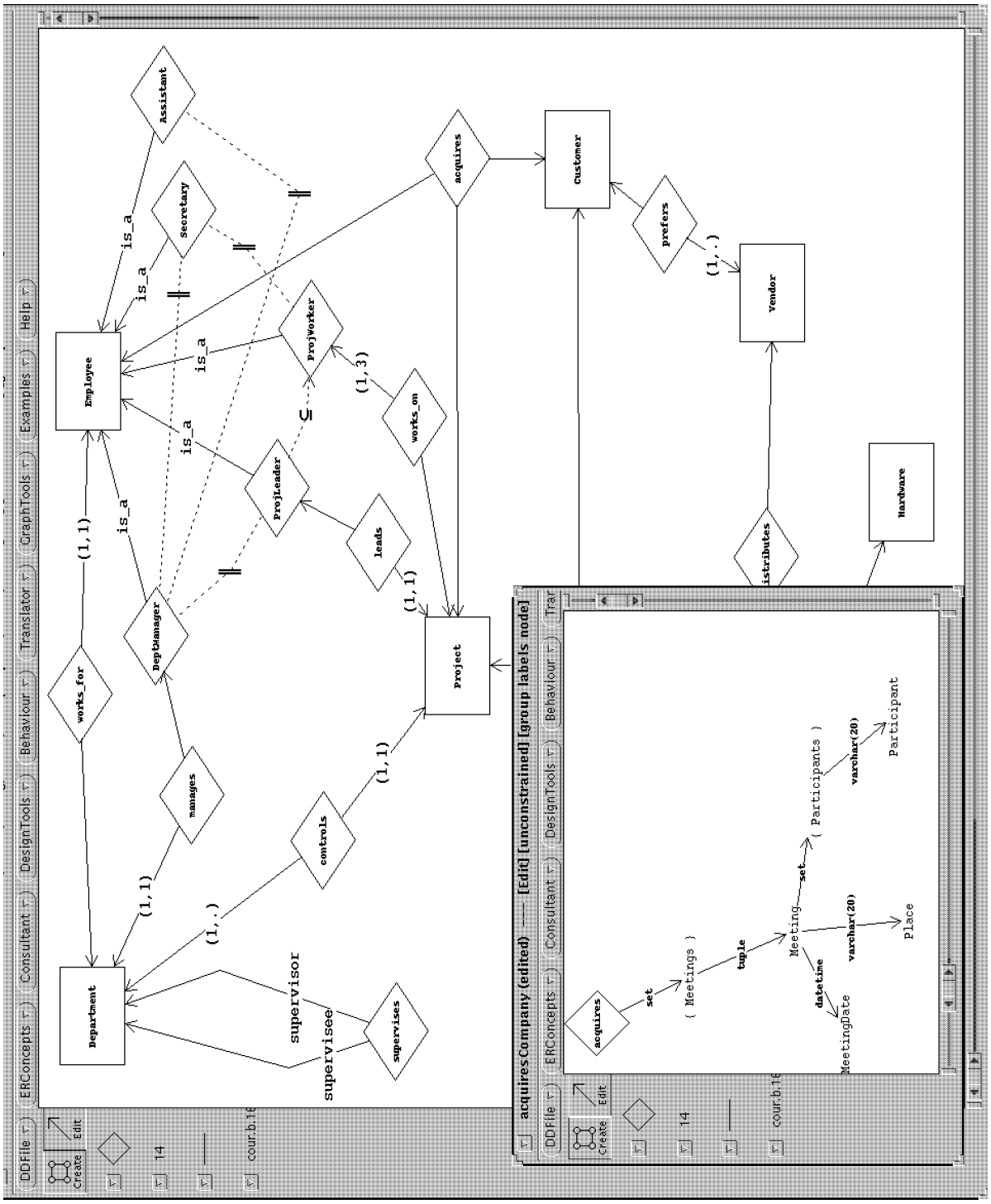


Figure 3.22: RADD Representation of the acquires Type, including the Attribute View on the acquires Type.

3.4 Summary and Outlook

The database management system concepts considered in this Chapter provide the maintenance of new-generation database applications. That is, structured data, control of automatical data updates, and data that is provided and accessed not only by single servers, or database client/server architectures, but also distributed and maintainable by networks, such as from HTML pages on the Web browser, or Java applets which are executed from the Web browser.

The data models of this Chapter provide the design of new-generation database applications. However, most of the data models and database design tools used to implement them have certain restrictions, which make the model not generally applicable. We think that the requirement to use only hierarchical databases is to restrictive.[18] More concretely, we think that a database design must be strictly hierarchical, but a database realization need not necessarily to use a fully hierarchical schema. But, most data models and database design tools of today are limited in the form that the database schema that is generated from these can only be fully hierarchical. The HERM and the RADD data model force the designer to construct a rigorous hierarchical conceptual database schema as well. However, they do not exclude that the internal schema which is used for the database implementation may be not completely hierarchical.

The "D" model of Section 3.3.2.4 is the most abstract data model presented here. It includes most of the new-generation database features. A disadvantage of the "D" data model is that it does not define a graphical notation. But, as we will see in Section 6.3, concepts of the RADD/raddstar data model and the conceptual specification language (CSL) can be compared to the concepts proposed for the "D" data model, although "D" has some features which are not included there and the data model of Section 6.3 has some properties which are not included or forbidden by the "D".

[18] Note, that even the result of the different relational database normalization approaches was a database implementation schema, that was not necessarily hierarchical.

Part II

Analysing Database Designs

> Database tuning is the activity of making a database run more quickly. "More quickly" usually means higher troughput, though it may mean lower response time for some applications.
>
> To make a system run more quickly, the database tuner may have to change the way applications are constructed, the data structures and parameters of a database system, the configuration of the operating system, or the hardware. The best database tuners therefore like to solve problems requiring broad knowledge of an application and of the computer system.
>
> Dennis E. Shasha,
> in the Preface of [Sha92].

Chapter 4

Database Optimization Scenarios

In Section 2.2.6, 2.3.5, and 3.3.2 we have shown some characteristics of database management systems concerning their logical data mapping and interface that they offer to the database administrator and application programmer. Now, in contrast to the data representation and integrity maintenance concepts which are typically used by the new-generation Web database tools which were described in Chapter 3, this Chapter concentrates on the logical and physical storage organization of typical database management system implementations, and associates them with database optimization scenarios.

The Chapter is organized as follows. In Section 4.1, we illustrate some examples for database optimization. Previous works of the author concerning transaction implementations did concentrate on redesigning database schemata and transactions for the purpose of making database states more consistent and transactions run quicker. These experiences as well as differences of the RADD/raddstar approach to previous database optimization approaches are described in Section 4.1.1. Section 4.1.2 decribes a formal approach to transaction performance tuning, the *Transaction Chopping* algorithm, which is found in [Sha92], Chapter 2, pp. 16-20, and Appendix A2. In Section 4.2, we give an illustrating example for optimization of a relational SQL database schema, which demonstrates how integrity maintenance and selection of the database schema that is used by the implementation influence each other. In this Section, we will give a short introduction to the reflection of the performance and behavior bottlenecks to the conceptual schema, which is realized by the RADD/raddstar. Section 4.3 gives a summary of the Chapter, and an outlook how we continue in the subsequent Chapters of Part II.

4.1 Database Optimization Scenarios

This Section exhibits on some database optimization scenarios. More scenarios are found in *Database Tuning* ([Sha92]) and the *Database Performance Tuning Handbook* ([Dun98]), from which ideas for this work have been won. Own experiences of the author are included.

Section 4.1.1 describes how database performance is improved by using different conceptual, logical, and physical data representations. Section 4.1.1 considers normalization and denormalization as well. As said in Section 1.3.1 and Section 2.3, normalization does not ever have advantages over denormalization, but in some situations normalization is also unadvantegeous. This is also mentioned in *Database Tuning* and the *Database Performance Tuning Handbook.* However, regarding this point we do not completely agree with the authors of *Database Tuning* and the *Database Performance Tuning Handbook,* while there are more situations than those mentioned, where denormalization is the necessary mean to tune an existing database system. Section 4.1.1 gives an introduction of this approach. Section 4.1.2 introduces the *Transaction Chopping* algorithm of *Database Tuning,* which makes transactions smaller for the purpose of increasing concurrency of database applications.

4.1.1 Conceptual, Logical, and Physical Data Representation

The same semantics of a modeling discourse (i.e., of the "mini world") can be expressed using different conceptual data models. Figure 4.1 shows that is generally possible to model the same mini world using different conceptual data models.

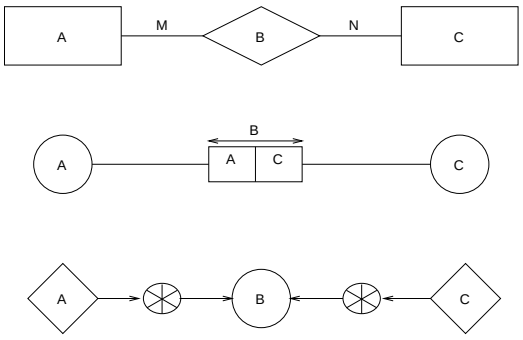


Figure 4.1: Entity-Relationship, NIAM, and IFO Representation of a binary many-to-many Association.

It should be clear that modeling the same semantics requires the more or the less effort, depending on the data model in use. To enlighten this statement, look at the more detailed data schemata (Attributes included) in Figure 4.2, which model the same real world entities, once by the ERM and once by the NIAM.

Implementation-oriented logical data models introduce most often concepts which are assumed valid for each member of the according DBMS class. As mentioned in the Introduction, Section 1.2, the *relational model of data* ([Cod70]) generally does not permit to use collection-typed attributes in sense of list, set, bag, and so on, because this violates

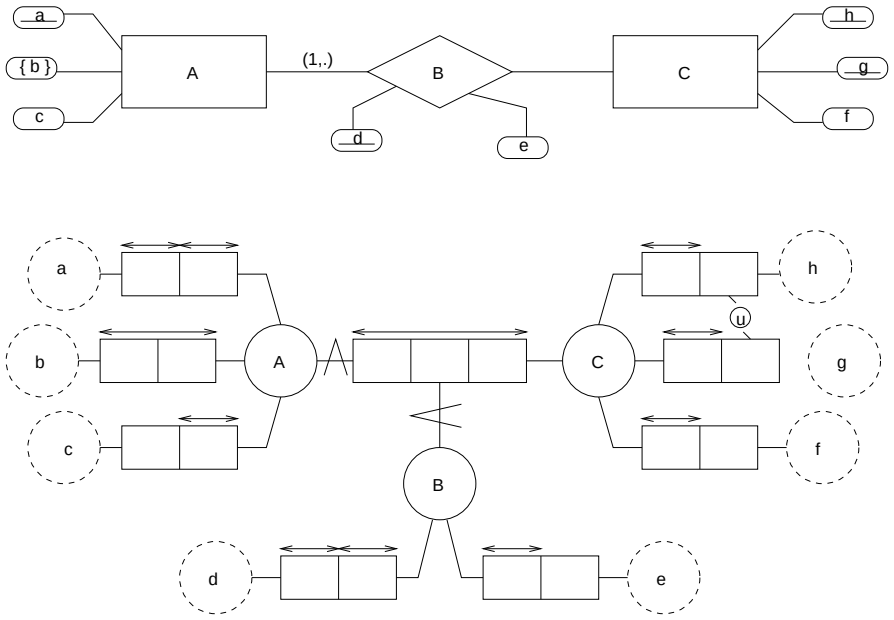


Figure 4.2: Entity-Relationship and NIAM Representation of the binary many-to-many Association (Attributes included).

the *first-normalform assumption* (1NF, refer to Section 2.3.2.1). Also, the new generation relational DBMSs, such as Oracle8, do not permit to introduce these types directly, but only by means of *tables* and *types* that are previously created. Therefore the relational data model(s) often introduces relations into a data schema, that are actually not "stand-alone" real world entities, but are used to represent collection-typed attributes. (Refer to [TDF86], the data model proposed there seems to make no differences from this viewpoint of the "relational world".)

4.1.1.1 Denormalization / Vertical Decomposition

Figure 4.3 represents the relational data schema for the ERM schema and NIAM schema in Figure 4.2.

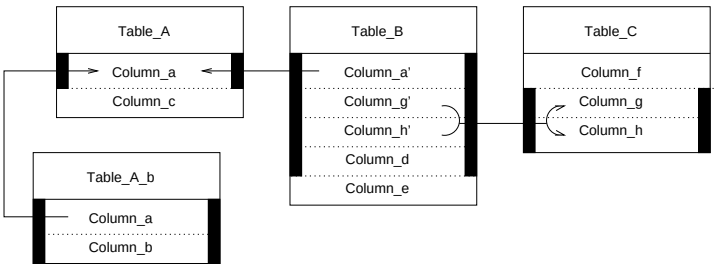


Figure 4.3: Relational Representation of the binary many-to-many Association.

Illustration. The schema of Figure 4.3 represents the collection-typed attribute $\{b\}$ which is represented by the circle labeled $\{b\}$ in the ERM schema and by the multivalued

fact between the nodes labeled A and b in the NIAM schema, by a separate relation, Table_A_b. The key of Table_A_b is its whole relation schema, namly the combination of (Column_a,Column_b). However, even if attribute $\{b\}$ of Figure 4.2 is an attribute where exactly one b-value for each a-value is the most common case, and no or more than one value in $\{b\}$ states an unfrequent exception, then it could also be reliable

1. to *denormalize* the schema, that is to represent $\{b\}$ as a repeated attribute $b1$,$b2$,... of the "parent" relation Table_A (such that the $b1$,$b2$,... are nullable, respectively),
2. or, to *vertically decompose* the schema (Column_a,Column_b,Column_c), such that we use a relation with schema (Column_a,Column_b,Column_c) holding the first b-value for each a-value, and a relaton with schema (Column_a,Column_b) holding the other b-values (those b-values for which an a-value has more than one b-value)

and not to represent this association by the separate "child" relation Table_A_b. This requires (1.) that applications need to transfer the flattened attribute $\{b\}$ to a set of values, or (2.) to use *union selects* to retrieve the data of this new mapping from the database. However, most applications transfer the data which they retrieve from the database as well, and *union selects* are generally much less costly than *joins*. These new structural mappings of the database relations, (1.) and (2.), improve select and insert, delete, update performance as well. (Refer also to [CG93].)

4.1.1.2 Schema Selection and Physical Database Structure

The latter consideration forces the fact that instantiation-related criteria must be strongly noticed, even if they could be used to simplify the database implementation schema, and so, the internal representation of the conceptual schema. As another example, let us consider the data schema mapping *bank accounts* in Figure 4.4.

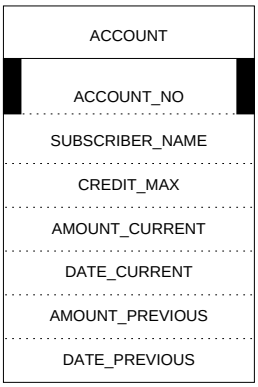


Figure 4.4: Relational Data Schema "Account".

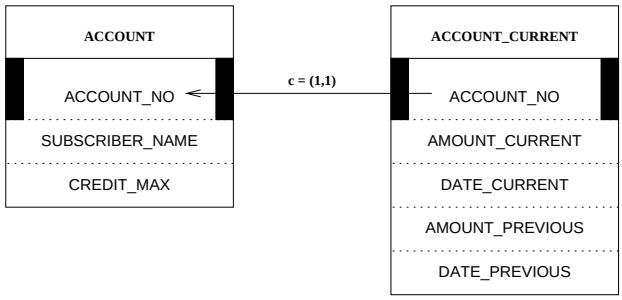


Figure 4.5: Alternative Relational Data Schema "Account".

This schema maps correctly the functional dependency ACCOUNT_NO → (SUBSCRIBER_NAME, CREDIT_MAX, AMOUNT_CURRENT, DATE_CURRENT, AMOUNT_PREVIOUS, DATE_PREVIOUS), but does it provide good performance ?

To examine this question, let us consider the data profile in Table 4.1, that presents the relative number of *selects, inserts, deletes, updates* on the relation ACCOUNT.

ACCOUNT	relative nr. of selects	relative nr. of inserts	relative nr. of deletes	relative nr. of updates
ACCOUNT_NO	2500	1	(1)	-
SUBSCRIBER_NAME	500	1	(1)	(1)
CREDIT_MAX	2500	1	(1)	6
AMOUNT_CURRENT	3000	1	(1)	1500
DATE_CURRENT	3000	1	(1)	1500
AMOUNT_PREVIOUS	3000	1	(1)	1500
DATE_PREVIOUS	3000	1	(1)	1500

Table 4.1: Access Profiles for the Data Structure "ACCOUNT".

The Table identifies that AMOUNT_CURRENT, DATE_CURRENT, AMOUNT_PREVIOUS and DATE_PREVIOUS are frequently updated. It seems therefore appropriate to hold the values of these attributes on a seperate relation such that the according items can be quickly accessed and updated. The schema according that fact is presented by Figure 4.5.

Data Clustering. Some database management systems, such as Oracle, allow to cluster two relations together, based on the key of one of the relations. The example which we used in Section 4.1.1.1 could alternatively be implemented using data clustering. The example of the current Section is also an application for data clustering. But, data clustering does not have advantages only. It has the advantage that queries on the cluster key are fast, but full table scans (the whole relation is retrieved) are somewhat slower, and also, inserts can cause overflow chaining. For more detailed information, see *Database Tuning,* Chapter 4.4.

References. There is an example similar the example of the current Section in [Sha92]. Additionally, data profile (access profile) considerations and their relation to optimization of database schemata are presented in [Su85]. Also, [Bom95] considers the according tuple numbers of data structures. As we will see, we need to consider both, relative operation frequencies and relative tuple numbers– and, we need to relate them to each other.

4.1.1.3 Schema Efficiency depends upon the Application Scenario

In this Section, we want to give an overview on the items which are necessary to perform database optimization ("database tuning"), and which of these items need to be considered by the RADD/raddstar.

Note that the subject of the thesis is

> *developing efficient conceptual database schemata; that is, to find the conceptual schema that provides an efficient or the most efficient implementation of the mini world.*

Schema efficiency can be considered as several properties:

(1) The schema implies as less as possible anomalies under consideration of operational maintenance. (duplicate keys, missing foreign-keys, a.s.o.)

(2) The Space occupied by the schema's objects is minimal.

(3) The complexity (Time) of the operations identified on the schema which are evaluated on basis of a chosen cost function is minimal.

(4) The (sum of the) operation complexities for a certain subset of often required operations which are identified on the schema is minimal.

(5) The set of operation complexities for a certain subset of often required operations which are identified on the schema is optimal under restriction that there is no runaway on the operation's complexity set that must be considered as a bottleneck.

All these aspects, respectively the proceedings that consider them (*normalization, denormalization, decomposition, clustering, ...*), have certain benefits. But, which aspect to consider more appropriately normally depends upon the use of the whole database system; that is, the database with its relations and the applications using the database. This way, schema efficiency is strongly depending on the application scenario.

4.1.1.4 Exhibit on the Schema Efficiency Properties

Consideration (1) specifies an argument that is traditionally, especially in relational database design, used to argue for *normalization*. Normalization aims to transforming a given database schema which is comprised of structures *Struc* and integrity constraints (semantics) *Sem*, to a new database schema $(Struc', Sem')$ such that the new schema maps the same mini world and redundancies are no longer contained in the new schema. Normalization is the typical viewpoint of traditional database management guides, traditional database design tools, and also picked up by textbooks like [Ull82, Win85, Eve88, RM92] and some recent database transformation proposals, e.g. [FV94, FV95].

Almost Valid Functional Dependencies. The envisioned target of consideration (2) mostly conincides with the target of consideration (1). But, although almost all normalization proposals refer to "lower storage complexity" it is often forgotten that exceptions for functional dependencies formally require to further normalize an existing database schema, but increase storage complexity as well. Assume, there is a relation schema $R = (A, B, C, D)$ which stores 10,000 tuples. However, 9,998 tuples fulfill the functional dependency $A \rightarrow (B, C, D)$, but 2 of the 10,000 tuples (only) fulfill $A \rightarrow (B, D)$ and $(A, B) \rightarrow C$. So, $A \rightarrow (B, C, D)$ doesn't hold for R. Thus, we would have to normalize R into $R_1 = (A, B, D)$ and $R_2 = (A, B, C)$. Assume (arbitrarily) each attribute, A, B, C, and D, has size 10. Then, the first database (with the non-normalized relation R) occupies 400,000 bytes, while the second database (with relations R_1, R_2) occupies 600,000 bytes.

Aspect (3) covers more appropriately such cases, in sense that now the operation complexity is considered. For example, for the above two database schemata ($\{R\}$ and $\{R_1,R_2\}$), to add one new (A, B, C, D) tuple to the database we must invoke only 1 *insert* operation for the first database schema, while we have to invoke 2 *insert* operations for the second database schema. Also, if we want to retrieve a certain item from the database, e.g. (A, B, C, D) WHERE A = 'my_name', then we can get this tuple directly from the first database, whereas we must perform a possibly "expensive" join $(R_1 \bowtie R_2)$ to retrieve the item from the second database.

Consideration (4) and (5) consider global and general optimization criteria for operation complexities. Consideration (5) takes into account that although the sum of operation complexities is minimal, there is not a certain operation– possibly an often required and very important operation –that is a runaway w.r.t. its complexity, and must therefore be considered as a bottleneck.

The ONF Assumption. Recall case (2) and (3). The wisdom of some lecture books that Space and Time are mutual exclusive (also mentioned in [Bom95]), is not necessarily correct. In general, there is no overall sentence to formally express the dependence of a Space and Time complexity in databases. The assumption of the NIAM / ORM researchers group is to provide an *optimal normalform* (ONF) schema whenever the data schema is in *fifth normalform* (5NF) and the number of relations is minimal. This was published in [LN88] and is found in many NIAM / ORM papers.

But, the considerations at the end of Paragraph Illustration in Section 4.1.1.1 deviate from such assumptions. Such assumptions have to be looked at as desires, since generally *third normalform* (3NF) is available for each relational database schema, but not BCNF, and therefore not 4NF and not 5NF. Absolutely in contrast to the ONF assumption is also the data profile example in Section 4.1.1.2, which shows that the higher relation number provides the schema which is the optimal according performance issues.

For these reasons, we must consider database implementation and population aspects as well. Implementation aspects are anomalies, referential dependencies, cost of join operations and pre-required operations or triggering actions. Population aspects are the uniformity of data, tuple numbers or correlations between them (relative tuple numbers), and criteria whether certain sets are changed frequently or not. These criteria are typically used for physical and logical database tuning after the problems of the running production database recognized. But, most often they are already known in advance of implementation, when conceptually designing the database. However, since knowing that a relational database management system (of today) does not support structured and nested domains (records, collection types) directly, these design criteria are often too early omitted, what disables safely extending the database schema whenever new requirements are specified.

4.1.2 Lock Tuning and Transaction Chopping

We have mentioned in Section 2.3.5.2 that locks can impact applications to behave inconsistently (because "no more locks available"). On the other hand, if they are set and not immediately released, they can create crucial performance bottlenecks, e.g. if a long-time transaction sets locks and other transactions are waiting for data which are locked by that transaction. In worst case– but that most often occurs only if the transaction and lock dependencies are incorrectly implemented –deadlocks are created.

4.1.2.1 Lock Tuning

According to these Scenarios, *Database Tuning* proposes the following measures to tune locking:

1. Eliminate locking when it is unnecessary.
2. Take advantage of transactional context to chop transactions into smaller parts.
3. Weaken isolation guarantees when the application allows it.
4. Use special system facilities for long reads.
5. Select the appropriate granularity of locking.
6. Change your data description data during quiet periods only. (Data Definition Language Statements are harmful.)
7. Think about partitioning.
8. Circumvent hot spots.
9. Tune the deadlock interval.

Furthermore, locking is unnecessary if only one transaction runs at a time, e.g. at initialization time or base data load time, or if all transactions are read-only. And, reducing

overhead by suppressing the acquisition of locks *may not be an enormous performance gain, but the gain it does provide should be exploited.*

4.1.2.2 Transaction Chopping

The purpose of *Transaction Chopping* is making transactions smaller– that is shorter – for the purpose of increasing available concurrency. Making transactions shorter has two effects on performance:

1. *The more locks a transaction requests, the more likely it is that it will have to wait for some other transaction to release the lock.*
2. *The longer a transaction T executes, the more time another transaction will have to wait if it is blocked by T.*

The *Transaction Chopping* algorithm uses simple graph theoretical ideas to break up transactions in a safe way, such that the different pieces have no longer control dependencies among themselves, can be executed in parallel. The assumptions for transaction chopping are as follows

1. All transactions that will run in some interval can be identified.
2. The goal is to achieve full isolation (degree 3, consistency).
   According the terms which we mentioned in Section 2.3.5.2, this is equivalent to:
   (a) The same data item x is read and/or modified by the same transaction T only (i.e., at most) once. A transaction must read a data item x, before it writes it. A data item x can have multiple "shared" locks (i.e., locks which are set by transactions for the purpose to read x only, but not to modify it), or otherwise, one "exclusive" lock (a lock which is set for the purpose to modify x; i.e., to write x).
   (b) It follows, that a transaction T which intends to modify some x must request an "exclusive" lock. This can be provided by the DBMS only if no "shared" lock is set on x. Otherwise, if a "shared" lock is set on x, T must wait for the "shared" lock (all "shared" locks) be released.
   (c) It follows furthermore, that no transaction reads data which are modified by other transactions at the same time ("no dirty reads"). Otherwise, if a transaction T needs to read a data item x which is modified by some other transaction at the same time (i.e., x is locked "exclusive" by the other transaction), T must wait for the "exclusive" lock on x be released.
   (d) Locks on read data are released by a modifying transaction T when T commits, but not before. (Otherwise, a data item x which was read by T, may already have been modified by some other transaction that completes before T, although the computations of T base on the x that did have a state different from that which it has in the database when T completes.)

(e) Pieces of a "long" transaction which are writing an object x (like $T11$, $T12$, and $T112$, shown in the examples below) send a `prepare to commit`,[1] by which the "exclusive" lock on x is released.

3. If a piece of a transaction which is chopped up makes one or more calls to rollback, and other pieces may have committed before, the database state must be the same as if the original transaction rollbacks.
4. Program variables which are not in the database, and which are modified by a transaction that completes but does not commit (rollbacks), must be in consitent states after transaction completion.
5. If a failure occurs, it is possible to identify which transactions completed before the failure and which ones did not.

The correctness of transaction choppings is characterized by two kinds of edges:

- *C edges (conflict). Two pieces p and p' from different original transactions are in conflict if there is at least one data item x that both access and at least one modifies.*
- *S edges (siblings). Two pieces p and p' are siblings if they come from the same transaction T.*

An SC-Cycle in a chopping graph identifies conflicts between transactions. That is, the transaction have then been broken up so far that possibly dead locks can occur.

Consider the following example of the transactions $T1$, $T2$, and $T3$, which perform read accesse (R) and write accesses (W, "modifications") on the data items x and y.

```
T1:   R(x) W(x) R(y) W(y)
T2:   R(x) W(x)
T3:   R(y) W(y)
```

Using these transactions, there is no lock conflict, because either $T1$ or $T2$ modify data item x without critical dependence on the order, and either $T1$ or $T3$ modify data item y without critical dependence on the order. However, the transactions $T1$, $T2$, and $T2$ can only be parallelized partially. That is:

- if $T1$ is started before $T2$, then $T2$ can not start until $T1$ completes (issuing a commit or rollback); so, $T2$ may wait for $T1$'s $W(y)$, although $T2$ does neither read nor write the data item y,
- if $T1$ is started before $T3$, then $T3$ can not start until $T1$ completes (issuing a commit or rollback),
- if $T3$ is started before $T1$, and then $T1$ (which must wait for $T3$ to complete, before it can do the $R(y)$) is started, $T2$ must wait until $T1$ completes,

[1] We presuppose that we are using a DBMS supporting the *Two-Phase Locking* (2PL) protocol.

- if $T2$ is started before $T1$, and then $T1$ (which must wait for $T2$ to complete) is started, $T3$ must wait for $T1$ to be completed; in this case, there is no parallel execution, such that the transaction execution is serialized.

We can break up $T1$ into $T11$ which reads and modifies x and $T12$ which reads and modifies y.

```
T11:  R(x) W(x)
T12:  R(y) W(y)
```

There is also no lock problem now, the chopping graph is shown in Figure 4.6.

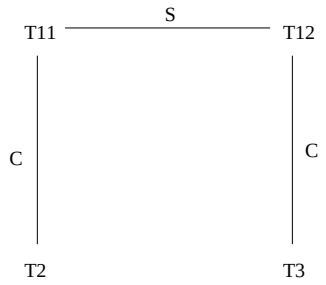


Figure 4.6: Chopping Graph without SC-Cycle.

But, now

- if $T2$ is started before $T11$, and then $T11$ (which must wait for $T2$ to complete) is started, $T3$ must not wait for $T11$ to be completed (but only, and only if started previously, for $T12$, which can be executed in parallel to $T11$); in this case, the parallel execution is possible, such that the transaction execution needs not be serialized, like in the latter case above.

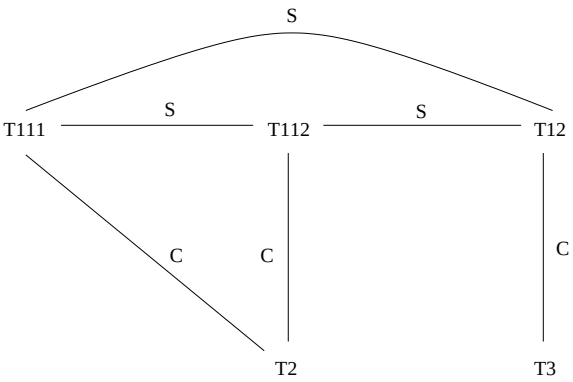


Figure 4.7: Chopping Graph with SC-Cycle.

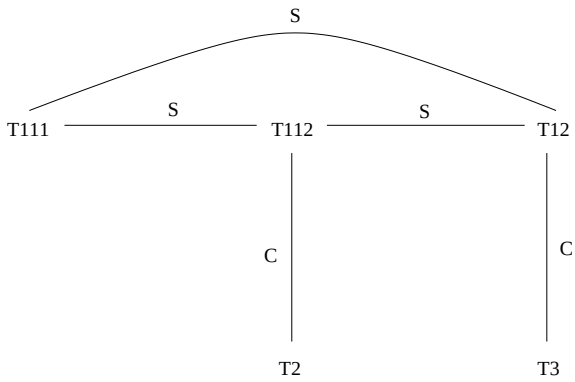


Figure 4.8: Chopping Graph without SC-Cycle.

We could further split $T11$ into $T111$ and $T112$.

```
T111: R(x)
T112: W(x)
```

Figure 4.7 shows the according chopping graph, which contains an SC-Cycle. (If $T111$, which does issue the write $W(x)$, issues $R(x)$ and after that $T(2)$ issues $R(x)$, then $T2$ waits for $T1$ (i.e., $T112$) to release the "shared" lock set by $T111$, and $T112$ waits for $T2$ to release the "shared" lock.)

By contrast, if $T2$ consists of $R(x)$ only

```
T1:   R(x) W(x) R(y) W(y)
T2:   R(x)
T3:   R(y) W(y)
```

we can break up $T1$ such that we get

```
T111: R(x)
T112: W(x)
T12:  R(y) W(y)
T2:   R(x)
T3:   R(y) W(y)
```

and there is no SC-Cycle. This is represented by Figure 4.8.

In *Database Tuning,* it is shown that an incorrect chopping can not be made correct again by further breaking up transactions. Furthermore, it presents and proves the *Transaction Chopping* algorithm which finds the optimal chopping. However, for reasons of space we can not present the algorithm and its prove here. So, the interested reader is directed to *Database Tuning,* Appendix A2 ([Sha92]).

4.2 Application Scenario: Conceptual Database Optimization based on Integrity Maintenance and Schema Transformation

Let us consider a part of the conceptual schema, its implementation by a relational database, and the implementation's optimization. The schema described by the following scenario is a subschema of the *Company* schema that we did already present.

A company is organized into deparments. A department has employees which are working for the department, such that one employee manages that department. An employee works for exactly one department. We keep track on the department's number (Dnumber), name, and locations. We keep track on the employee's social security number (Ssn), birthdate, name, address, sex, and salary. The name of an employee consists of a sequence of

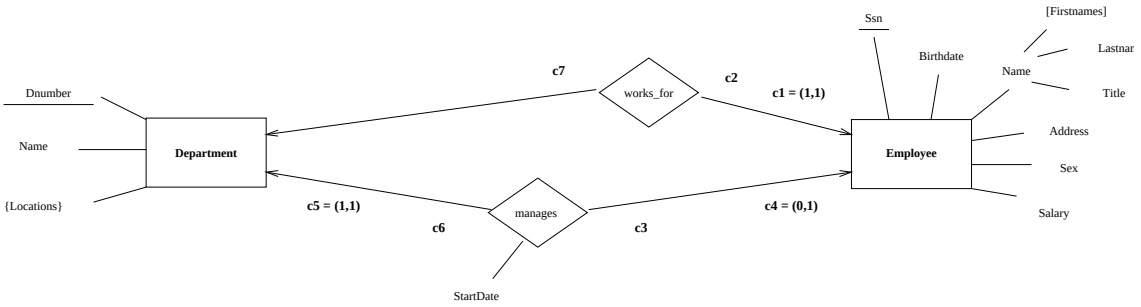


Figure 4.9: Department-manages-Employee-works_for-Part of the Company Schema.

firstnames, a lastname, and eventually a title. We keep further track on the start date the employee who is the department manager begins managing that department.

Figure 4.9 shows the HERM entity-relationship schema describing that scenario. The schema contains the entity types *Employee* and *Department* and the relationship types $works\_for$ and *manages*. For illustration purposes, we have labeled the cardinality constraints and arrows (references, referential constraints) here:

(c1) the cardinality constraint on the reference ($works\_for$,*Employee*),
(c2) the reference $works\_for \rightarrow Employee$,
(c3) the reference $manages \rightarrow Employee$,
(c4) the cardinality constraint on the reference (*manages*,*Employee*),
(c5) the cardinality constraint on the reference (*manages*,*Department*),
(c6) the reference $manages \rightarrow Department$, and
(c7) the reference $works\_for \rightarrow Department$.

The description of the scenario specifies 1:1 associations between *Employee* and $works\_for$, and *Department* and *manages*. These are graphically represented by the references and the (1,1)-cardinality constraints which are labeled (c1,c2) and (c5,c6), respectively, such that the conceptual schema suggests to group the according types when implementing the database. Following that suggestion, we get a schema with the grouped structures (Employee,works_for) and (Department,manages).

Figure 4.10 shows how the relational database would be implemented in SQL according these design informations:

- Paragraph (1) shows how the database schema is realized when relating strictly to the demands of the conceptual schema (Figure 4.9). One can easily recognize that, whenever implementing the schema that way, a tuple can be neither inserted into the *Emp* table, which corresponds to the (Employee,works_for) structure, nor into

```
(1)
 CREATE TABLE Dept
 ( Dnumber      DECIMAL(6,2) PRIMARY KEY,
   Name         CHAR(20) NOT NULL,
   Locations    VARCHAR(60) NOT NULL,
   MgrSsn       DECIMAL(9) NOT NULL,
   MgrStartDate DATE
 ) ;

 CREATE TABLE Emp
 ( Ssn          DECIMAL(9) NOT NULL PRIMARY KEY,
   Birthdate    DATE NOT NULL,
   Sex          CHAR(1) NOT NULL
     CHECK (Sex IN ('m','f')),
   Firstnames   VARCHAR(30) NOT NULL,
   Lastname     VARCHAR(20) NOT NULL,
   Title        VARCHAR(10),
   Address      VARCHAR(40),
   Salary       FLOAT CHECK (Salary > 5000),
   DeptNum      DECIMAL(6,2) NOT NULL
     CONSTRAINT emp_dept REFERENCES Dept
 ) ;

 ALTER TABLE Dept ADD CONSTRAINT dept_mgr
     FOREIGN KEY (MgrSsn) REFERENCES Emp ;
```

```
(2)

(2a)
 ALTER TABLE Emp MODIFY DeptNum NULL ;

(2b)
 ALTER TABLE Emp DISABLE CONSTRAINT emp_dept ;

(2c)
 ALTER TABLE Emp DROP CONSTRAINT emp_dept ;

 CREATE TRIGGER emp_dept
  AFTER INSERT OR UPDATE ON Emp
  FOR EACH ROW
   WHEN ( NOT NEW.DeptNum IN
               (SELECT Dnumber FROM Dept) )
  BEGIN
   INSERT INTO Dept
     (Dnumber,Name,Locations,MgrSsn)
    VALUES ( NEW.DeptNum,
             concat(NEW.Lastname,"'s Dept"),
             concat(TO_CHAR(NEW.DeptNum),"'s Loc"),
             NEW.Ssn ) ;
  END ;
```

Figure 4.10: SQL-Commands for Creation and Repair of the Database.

the *Dept* table, which corresponds to the (Department,manages) structure. For illustration purposes, we have included the representation of this implementation using entity-relationship modeling concepts in Figure 4.11.[2]

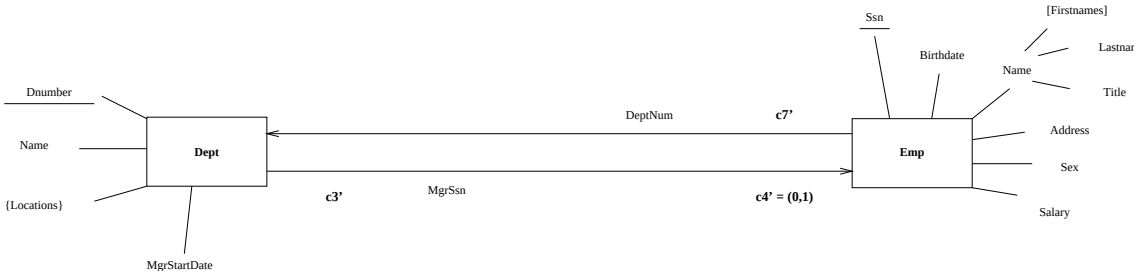


Figure 4.11: Internal Schema (Physical Schema).

- Paragraph (2) shows how the database schema can be repaired. The alter statement of (2a) drops *Emp.DeptNum*'s NOT NULL constraint. Alternatively, (2b) can be used to disable the referential constraint *emp_dept*. It can be re-enabled after the Department managers and the Departments have been inserted. Paragraph (2c) shows how the referential constraint *emp_dept* is substituted by a trigger. The trigger generates automatically the Department as soon as an Employee *emp* is

[2] Although using entity types and relationship types this schema is not considered a conceptual schema (a HERM respectively RADD schema, refer to the description of the properties of a HERM schema in Section 3.3.3.1), since it models cyclic referential dependencies between the structures *Emp* and *Dept*.

inserted such that *emp.DeptNum* doesn't already exist in *Dept.Dnumber*. Until now, the combination of (1) and (2c) could be seen as the best choice.

- Paragraph (3) in Figure 4.12 continues on the create and alter statements of (1) and the changes made by the alter table and create trigger statement of (2c). (3a) adds a constraint that causes automatical delete of Employees (*Emp*) whose Department (*Dept*) has been deleted. (3b) adds a trigger that ensures that the inserted or updated Employee who becomes a Department manager does really manage only one Department (constraint c4 in Figure 4.9). Paragraph (3c) shows how the *Dept* and *Emp* tables are restructured. The $MgrSsn$ and $MgrStartDate$ columns (attributes) are removed from the *Dept* table and $MgrStartDate$ is added as an optional attribute to the *Emp* table. Then, the view *Department* is created to substitute the old *Dept* table. The integrity maintenance w.r.t. Department managers is now controlled by the new trigger *emp_dept* which ensures that the first Employee who is inserted to work for the new Department is considered as the Department's manager. Also, by the trigger there can be no other Employee of the same Department for which $Emp.MgrStartDate$ is NOT NULL, such that he is considered as the Department's manager.
- The latter SQL database schema is not represented anymore by the conceptual schema of Figure 4.9. Furthermore, the actions (2c) and (3c) gently presuppose that the Department manager must work for the same Department, which was not represented by the first conceptual schema– although this information was already included in the scenario description. We need to mention, that in real life one also can often observe that design informations are acquired and verified again while or after the database schema is installed.

Hence, according the database schema resulting from (1), (2c), and (3c), the conceptual schema should be modified such that it is easier to find a correct and efficient implementation.

4.2.1 Repairing the incomplete Database Design

It is clear that correctness and efficiency of database operations, like select, insert, delete, and update in SQL, often can be improved by using internal schemata which differ from the given conceptual or logical database schema. These aspects take especially then place when complex transactions reveal performance and/or consistency problems. Internal design considers the concrete operations and transactions more appropriately in terms of behavior, generated transaction results, and performance, such that, like illustrated in Figure 1.2 and shown by the optimization scenario of this Section, the internal design can repair conceptual and logical database design mistakes, after it recognizes them.

```
(3)

(3a)
 ALTER TABLE Emp
  ADD CONSTRAINT emp_dept_del
     FOREIGN KEY (DeptNum)
       REFERENCES Dept ON DELETE CASCADE ;

(3b)
 CREATE TRIGGER emp_1mgr
  AFTER INSERT OR UPDATE ON Dept
  FOR EACH ROW
   WHEN ( (SELECT count(*) FROM Dept
             WHERE MgrSsn = NEW.MgrSsn) > 1)
  BEGIN
    raise_application_error( -20477,
     'A Department Manager must not manage
       more than one Department !' ) ;
  END;

(3c)
 DROP TRIGGER emp_dept ;

 ALTER TABLE Dept
     REMOVE ( MgrSsn, MgrStartDate ) ;
```

```
(3c, continued)
 ALTER TABLE Emp ADD MgrStartDate DATE ;

 CREATE VIEW Department AS
  SELECT
   Dept.*,Emp.Ssn MgrSsn,Emp.MgrStartDate
    FROM Dept,Emp
   WHERE Dept.Dnumber=Emp.DeptNum
     AND Emp.MgrStartDate IS NOT NULL ;

 CREATE TRIGGER emp_dept
  AFTER INSERT OR DELETE OR UPDATE ON Emp
  BEGIN
   IF (SELECT DeptNum,count(*) FROM Emp
        WHERE MgrStartDate IS NOT NULL
        GROUP BY DeptNum HAVING count(*)<>1)
   THEN
    raise_application_error( -20478,
     'Violated Constraint:
       Each Department must have exactly one
       Manager in the Employee Relation !' ) ;
   END IF ;
  END ;
```

Figure 4.12: SQL-Commands for Optimization of the Database.

Points of view how to make database applications run more quickly have already been outlined by different authors [Gil91, Sha92, CG93]. In addition, DBMS manuals [Syb93, PLSQL95] give guidelines how to improve database performance. In Section 4.1 we have explained RADD/raddstar's and *Database Tuning's* ([Sha92]) approach to database optimization.

Hence, mistakes which are made in early database modeling phases can be remedied by later phases. This kind of design repair can be done either by

1. selecting a different schema in advance of implementation, or else, by
2. restructuring the schema of the database that is already running.

But, as mentioned in Chapter 1, these approaches are often difficult to handle because they require changes to the database such that the changed internal views may not be represented externally anymore, and vice versa. This way, modifying or extending the database requires to repeat the whole design process once more.

Therefore, a better strategy is to make the designer aware about possible mistakes which he makes during information acquisition and conceptual design. Thus, we have to find a way back to reason about performance and consistency problems of the internal database schema, in order to apply conceptual database design optimization.

4.2.2 Optimizing the Example Schema

Let us recall the conceptual schema of Figure 4.9, and its internal schema implementation and optimization, which were presented by the SQL-code in Figure 4.10 and Figure 4.12.

On basis of the given schema which is represented by the SQL-code in paragraph (1) of Figure 4.10, we must additionally generate (remove) Employee (Department) items on insert (delete) of Department (Employee). Under these circumstances, the transaction contents and the costs of *insert into/delete from Employee*, *insert into/delete from works_for*, *insert into/delete from Department*, and *insert into/delete from manages* can be assumed high, such that these operations could be considered as operational bottlenecks. This is caused by the additional operations which are required by the applications or automatically generated by the database procedures and triggers, whenever the mentioned operations are invoked.

```
add conceptual optimization rule:
 when bottleneck(delete,s1) and bottleneck(delete,s2)
  and entity s1 and entity s2
  and exists s3,s4:
       ( (dcycle [s1,s3,s2,s4] or dcycle [s1,s4,s2,s3])
     and compatible [s3,s4] )
 do
  separate (group (s3,s4) (.,.)) [s4]
```

Figure 4.13: Specification of a Conceptual Schema Optimization Rule.

This set of (additional) operations may result in operation cycles, especially on deletions. That is, the whole database can be made empty by an delete operation that is invoked. To remedy this drawback, let us consider the conceptual schema optimization rule which is defined in Figure 4.13.[3] The rule specifies, that whenever the delete is an operational bottleneck for structure $s1$ and structure $s2$ of the current conceptual schema, respectively, and these bottlenecks result from a "delete-cycle":

- delete $s1$ invokes delete $s3$ (assuming, that $s3$ has a reference to $s1$), delete $s3$ invokes delete $s2$, delete $s2$ invokes delete $s4$ (assuming, that $s4$ has a reference to $s2$), and delete $s4$ invokes again delete $s1$
  – or –
- delete $s1$ invokes delete $s4$ (assuming, that $s4$ has a reference to $s1$), delete $s4$ invokes delete $s2$, delete $s2$ invokes delete $s3$ (assuming, that $s3$ has a reference to $s2$), and delete $s3$ invokes again delete $s1$

then the structures which have references to the entity structures, $s3$ and $s4$, must be inspected. If the structures $s3$ and $s4$ are "compatible" then they can be grouped to one structure, such that subsequently the one which has been grouped to the other is

[3] A syntactically different version of this rule is found in [Ste95] and [Ste96]. However, this version of the optimization rule is compatible with the CSL specification language of the RADD/raddstar, that we will present in Chapter 7.

extracted from the other again. So, the new extracted structure ($s4$) has only a reference to the other new structure (which was constructed by grouping the original $s3$ and $s4$), but no longer to the entity structures $s1$ and $s2$.

Note: *The rule specified in Figure 4.13 is not only restricted to direct references between structures, and therefore immediate invocations of deletes. The sfdcycle function considers also transitive invocations of deletes, such that transitive references are considered as well. E.g., we could have a path* s1 ¡- s3 ¡- s5 -¿ s2 ¡- s4 -¿ s1 *with high delete complexities for delete* $s3$ *and delete* $s4$ *such that the rule applies and generates* s1 ¡- (s3,s4) ¡- s5 -¿ s1 .

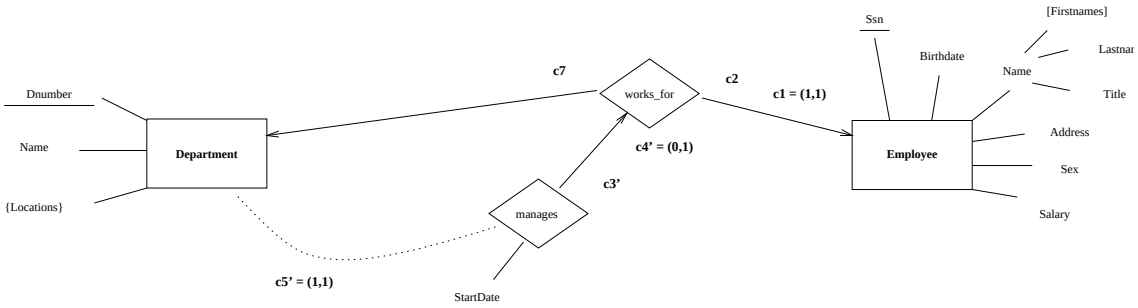


Figure 4.14: Optimized Conceptual Schema.

In this example, the *manages* and *works_for* relationship structures are compatible such that they can be grouped. This generates a new structure with an optional (nullable) attribute *StartDate*. Subsequently, the conceptual database optimizer re-generates the old relationship structures from this new structure. In the optimization of Figure 4.9, this results in the extraction of *manages* with its original attribute set ($\{StartDate\}$) from the grouped structure (works_for,manages), such that the optimized conceptual schema looks like shown in Figure 4.14. To preserve the semantics of the given schema of Figure 4.9, we need to add the following constraints to the new schema:

(c3') a new reference $manages \rightarrow works\_for$,

(c4') $card(manages, works\_for) = (0, 1)$, and

(c5') $card(manages, Department) = (1, 1)$.

The adding of these constraints must be automatically done by the conceptual database design optimizer. We refer to the new constraints of the optimized conceptual schema as c3', c4', and c5'– as they present the modified constraints c3, c4, and c5 of Figure 4.9. (The conceptual database optimizer has to retain the structures and connections between the structures, which were given for the original schema.)

The optimized conceptual schema (Figure 4.14) and the new constraints c3', c4', and c5' correspond to the implemented database schema that has been generated by the SQL-Commands of paragraph (1), (2c), and (3c). The referential constraints of *manages*– the references of *manages* to *Department* (c6) and to *Employee* (c3)– appear now as the reference from *manages* to *works_for* (c3').

4.2.3 Summary

In this scenario, it is not hard to see that:

1. The new schema provides a consistent and efficient implementation (new tuples can always be inserted into the Department relation).

2. And more important: the new schema models the mini world completely.
   (The Employee who manages the Department also works for that Department!)

4.3 Summary and Outlook

The Chapter presents some database optimization scenarios and shows how criteria for database optimization are recognized and won. Furthermore, the Chapter introduces how transactions and transaction costs are evaluated by the RADD/raddstar.

Section 4.1 described the presuppositions of the conceptual schema optimization approach of the RADD/raddstar. In Section 4.1.1 we described some database optimization scenarios. A reference according lock and transaction tuning (*Transaction Chopping,* [Sha92]) has been presented in Section 4.1.2. We consider these transaction locking concepts by the cost model of the RADD/raddstar, which adds additional terms to the costs of the transactions whenever it recognizes that the nesting is too deep.

In Section 4.2, we gave a short introduction to the reflection of the performance and behavior of the internal schema bottlenecks to the conceptual schema. We showed that it does not necessarily become necessary to optimize the internal database schema, since we are able to optimize the conceptual database schema during design.

The optimization scenario of Section 4.2 was presented by the author in [Ste96]. The scenario is here adapted considering the changes which have meanwhile been made according

- the transformation operations & transaction evaluations (Chapter 5),
- the compilation kernel (Chapter 6),
- and the conceptual specification language (CSL, Chapter 7)

of the RADD/raddstar.

Chapter 5

Integrity Maintenance, Conceptual Schema Mapping, and Fitness Evaluation

From the database optimization scenarios of Chapter 4 we obtain the cost model that has been realized in the RADD/raddstar, and evaluate the fitness of database operations that are used by the physical structures of the database ("physical costs"). That we consider the physical cost terms is for reason that the fitness evaluation for the conceptual database schema consists of

1. a conceptual schema to internal schema transformation,
2. the evaluation of transactions on the internal schema,
3. the estimation of the internal transactions' operational fitness,
4. and the reflection of the internal transactions' contents and costs to transactions and operation costs which are presented to the conceptual database designer.

The Chapter presents the specification, transformation, and optimization terms as well as the cost model that have been realized in the RADD/raddstar system.

The Chapter is organized as follows. Section 5.1 gives an overview on the dependencies between integrity maintenance and schema transformations. In Section 5.2, we show the predefined schema transformation operations of the RADD/raddstar. Section 5.3 describes the cost model and how the reflection of the operation costs and detected bottlenecks which are evaluated for the internal schema, to the conceptual schema is implemented. The conceptual bottleneck representation makes the database designer aware about possible design untidynesses and problems that may occur at time of the later database maintenance. Section 5.4 gives a summary of the Chapter, and an outlook on the usage of the presented fitness evaluation concepts in the RADD/raddstar.

5.1 Integrity Maintenance and Schema Transformation

The structure of the database schema, and the integrity maintenance and behavior of running transactions, on the other hand, are connected to each other, such that the choice of the database schema influences the contents of the transactions, and in turn, the behavior of the transactions can influence the choice of the database schema.

As we have mentioned in Section 3.3.2, Section 4.1.2, and Section 4.2, problems according integrity maintenance of the internal schema, e.g. preserving key and foreign-key dependencies by the implemented operation sequences of transactions, or controlling integrity constraints by rules (triggers), may not make the internal database schema efficient. E.g., it may raise lock scheduling errors, and so, not allow fast analysis of queries and updates such that the cost for preparing queries and updates is high. These situations occur, if the internal schema's structuring is not transparent to the DBMS and the time consume for analysis of the dependencies is high. (Refer to [Gün95].)

Therefore we propose, and have realized in the RADD/raddstar, another approach which makes the internal schema that is used for the evaluation of the conceptual schema's fitness simpler according the structures, their dependencies, and the integrity constraints remaining in the internal database schema.

5.1.1 Error Prevention Properties

As mentioned by the database optimization scenario of Section 4.2, the RADD workbench (the RADD/raddstar) presupposes error prevention properties for the transactional behavior in case of integrity violation. The error prevention properties can be expressed by means of behavior options, and may be specified more detailed by behavior rules that can include function calls. The behavior options are specified in RADD/raddstar a little bit similar the Codasyl specifications of *insertion* and *retention* options, shown in Section 2.2.5, such that they provide a simple specification interface for the database designer. However, instead of using insertion and retention options only, we provide "insertionOption", "deletionOption", "updateChiOption", and "updateParOption", and cascading behavior can be specified not only for deletes (like in Codasyl and today's SQL), but also for inserts and updates. The behavior options are related to references of the structures, e.g. R_1 which has a reference to R_2, such that R_2 is the parent structure (e.g. an entity type) and R_1 is the child structure (e.g. a relationship type), or, to cardinality constraints (CCs), functional dependencies (FDs), inclusion dependencies (IDs), and exclusion dependencies (EDs).

The term insertionOption specifies what to do, if a new record (or object) is inserted into the set of R_1 (${R_1}^t$, called occurence set of R_1– refer to Section 5.3.1.1) and the

referenced record does not (already) exist in ${R_2}^t$. The term deletionOption specifies what to do with the records in ${R_1}^t$ for which the referenced record is deleted from ${R_2}^t$. Since we do consider update operations as well, we have included "updateChiOption" and "updateParOption". Hence, in a value-oriented, equality and set-semantics based database– i.e. a relational database with keys as unique values in the according column(s) of a table, and foreign-keys which are equal to some key-value(s) of the referenced table –, updateChiOption specifies what to do, if a record of ${R_1}^t$ is updated such that it has a new foreign-key, and the foreign-key is not (already) key in ${R_2}^t$. The term updateParOption specifies what to do, if the key of some record in ${R_2}^t$ is updated such that there are now records in ${R_1}^t$, which have foreign-keys that have no more equal keys in ${R_2}^t$.

The behavior options of the RADD/raddstar can be set to the following values, which informally specify the behavior of the running database's transactions:

1. Restrict. Cancellation of the transaction as soon as any data inconsistency appears; that is, the transaction is aborted (rolled back) such that the complete old database state, as it was before invoking the transaction, is restored.
2. Cascade. This means *invoking repairing actions* as soon as data inconsistencies appear; e.g., if– on an insert operation –the referenced record (object) is missing in the parent structure's occurence set, then it is generated as well, or, if– on a delete operation –a record (object) is deleted from the parent structure's occurence set, for which still references from the child structure's occurence set exist, then these records (objects)– whose reference was deleted –are deleted as well.
3. Set Null. If some record (object) of the parent structure's occurence set, for which still referencing child records (objects) in the child structure's occurence set exist, is deleted, then the corresponding child references are set to "null" (in case that they are allowed to have null references; if they are not allowed to have null references, the transaction is rolled back).
4. Set Default. On insert of some new record (object) into the child structure's occurence set, a default record (object) of the parent structure's occurence set is used as reference from the new record. This default record in the parent structure's occurence set must never be deleted. If a record (object) of the parent structure's occurence set, for which still child records exist is deleted, then the references of the corresponding child records are set to the default record in the parent structure's occurence set.

The `set null` and `set default` options are usually used in cases where either referential values are not known, or else, referenced values are deleted from the target set of a referential constraint. They may also be used, if the according user interfaces do only allow to input data for a certain subset of the relation's attributes.

5.1.2 When do Transformations take place?

During the transformation process, the (transformation rule/integrity constraint)-pair that fires is checked against all modeled and implicite constraints and behavior rules. Then, a decision procedure determines whether the rule/contraint-pair fires actually, i.e. whether the transformation is applied or not. E.g., if the database designer explicitely specifies `on insert cascade` rules or models nullable references in the graphical RADD design, then a transformation action, like those which are given for the 1:1-associations (Employee,works_for) and (Department,manages) of the Company Schema, respectively, is not mandatorily applied. For example, the rule

For Ref works_for → Employee : on insert cascade

implies that the default transformation rule which is given for the constraints $works\_for \rightarrow Employee$ and $card(works\_for, Employee) = (1, 1)$ is not applied, because items of *Employee* and $works\_for$ will be inserted into the database the same time– for reason of the *cascade* rule which we assume to be implemented by a trigger.

This way, we consider

1. the insert operation for *Employee* to include the insert operation for $works\_for$,
2. or else, the insert operation for *Employee* to be not autonomous, but always triggered by the insert operation for $works\_for$.

In both cases, we can presuppose that the related *Employee* and $works\_for$ records are inserted into the database the same time, which is done by an database application.

```
General:
 on insert cascade,
 on delete cascade,
 on update parent do
  if nullable(child) then set null else set default fi,
 on update child cascade;
```

```
Special:
 on insert Department {Department=d} do
  if not (d in manages->Department)
   then insert manages {Department=d,...} fi,
 for CC(manages,Department) is (.,.) :
  on delete restrict;
```

Figure 5.1: General and Special Behavior Rules.

5.1.3 General and Special Integrity Maintenance Rules

Figure 5.1 shows example specifications for general and special behavior rules of the Company Schema. Note, that by the behavior options (restrict, cascade, set null, set default) we do not necessarily require the conceptual designer to describe explicitely the actions that are needed to maintain the integrity constraints of the database which is implemented for the Company Schema. That is, to specify or implement the concrete application code, and therefore, to go into details of the database and application realization– although it is possible to derive the database schema implementation and the application procedure code in a straight forward manner from the integrity rules.

5.2 Schema Transformation Operations

Since transactions must preserve integrity constraints which are designed for the database– either by key and foreign-keys (respectively references between structures), or else, by database rules (triggers) and procedures, which, again, must be correctly specified to implement the designed integrity constraint(s) correctly –we have to retain the integrity constraints of the conceptual database design during conceptual schema to the internal schema transformation, although they may be represented another way on the internal schema. As we have mentioned in Section 4.2 and 1.4, we preserve the original integrity constraints of the conceptual schema during optimization as well. This way, we can use the same operations for conceptual schema to internal schema transformation, and for conceptual schema optimization.

Let us consider what are in general the impacts of schema transformation to integrity maintenance.

5.2.1 Impact of Transformation to Integrity Maintenance

The transformation of the structures and integrity constraints, between the conceptual and the internal schema, respectively, has the effect that

- collapsing of structures leads to drop of structure references and integrity constraints, and
- separating of structures (e.g., for reason to maintain static and dynamic data of the same conceptual structure separately, or, to represent collection-typed attributes of the conceptual schema by a hierarchical or relational database schema) leads to adding new references and integrity constraints.

Thus, the internal representation of the conceptual schema is defined by a set of new structures and a set of new constraints. To keep track which transformations we are making, and which are the structures from which the new internal structures are generated, we use special pointers from the new structures to the old structures. These pointers are called *preceders*.

5.2.1.1 Preceders

If a new structure Emp is generated by grouping $Employee$ and $works\_for$, then $Employee$ and $works\_for$ are considered as preceders of Emp. Integrity constraints (references, cardinality constraints, and keys, FDs, IDs, and EDs) do have preceders as well, such that we can determine from which constraint a new constraint has been generated, for the internal schema and the optimized conceptual schema, respectively. So, if we assume an

operational bottleneck, then it is always possible for us to determine which graphically modeled or specified constraint of the conceptual schema is responsible for the bottleneck.

The constraints of the set of new constraints have to be considered only when the database system is implemented and transactions are running. So, they are considered only when valuating the given database design. Furthermore, we use conceptual and internal *transaction graphs* to provide a mapping between the internal and conceptual transaction contents, and, to evaluate the costs of the transactions which we present to the conceptual database designer. (The mapping will be presented in Section 5.3.3.)

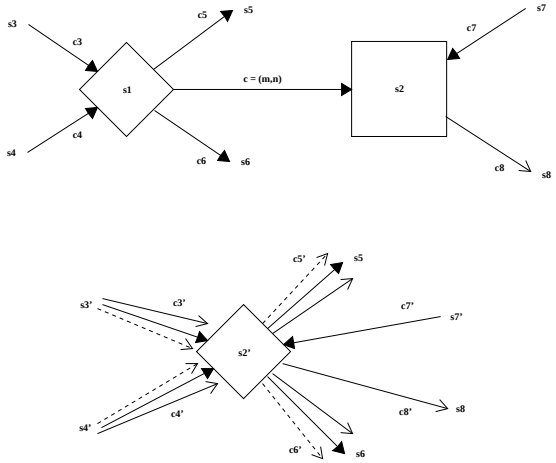


Figure 5.2: group (s1,s2) (m,n)

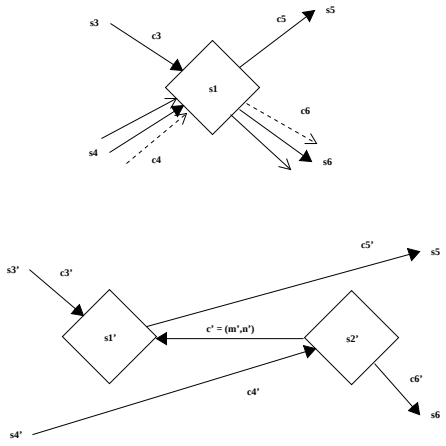


Figure 5.3: separate s1 [s2]

5.2.2 Basic Schema Transformation Operations

The RADD/*raddstar* schema transformer and optimizer provide five basic transformation operations: *group, separate, nest, unnest,* and *clusterize*. These reflect transformations like generation of repeating groups (group), extracting structures from other structures (separate), nesting of structures into other structures (nest), unnesting of structured or collection-typed attributes (unnest), and generation of unions, which are clusters in the RADD and HERM data model (clusterize). Additionally, the database designer has the opportunity to define own schema transformation operations using the *conceptual specification language* (CSL).

The basic transformation operations are presented in Figure 5.2, 5.3, 5.4, 5.5, and 5.6.

Note: *For reason to concentrate on the important aspects of the schema transformation operations, we have ommited the labels for the referential dependencies (references) in these Figures.*

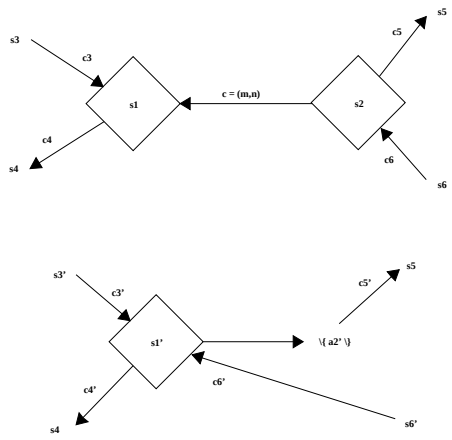


Figure 5.4: nest (s1,s2) tset

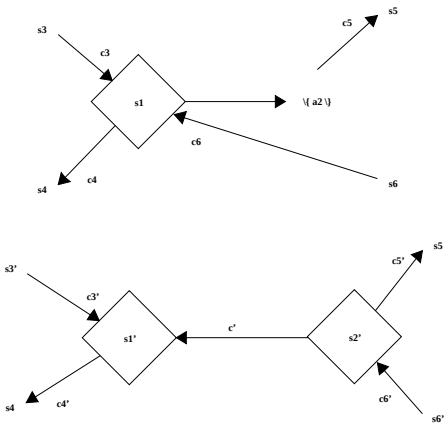


Figure 5.5: unnest s1 {a2}

The meaning of the transformation operations is as follows:

group (s1,s2) (m,n). The group operation collapses two structures ($s1$,$s2$) to one internal structure ($s2'$). A negative m or n, where m is not equal to -1 (-1 encodes "dot", or unknown) encodes that the group operation has to specialize to nest or clusterize. (If m is -1 then m as treated as if it were 0.) If group specializes to nest or clusterize, then a procedure evaluates the specialized transformation operation and the parameters of the specialized transformation operation which uses the the *plau_card*() function (refer to Section 5.3.2.4). Otherwise, each object in $s2^t$ has at least m and at most n objects in $s1^t$, a structured (record-typed) attribute of $s1$ is repeated m times mandatory *(not null)* and $n - m$ times optional *(with null)* in the new structure $s2'$. Also, all references to the previous $s1$ and from the previous $s1$ are replicated m times mandatory and $n - m$ times optional. For instance, as Figure 5.2 shows, if $m = 2$ and $n = 3$ then two not null arrows and one with null arrow, to and from the new structure $s2'$ are generated, respectively.

The group operation includes transformation of all integrity contraints which did reference a structure that is transformed. These are the cardinality constraints $c3$, $c4$, $c5$, $c6$, $c7$, and $c8$ (in Figure 5.2, which are transformed to $c3'$, $c4'$, $c5'$, $c6'$, $c7'$, and $c8'$, respectively. The group operation includes transformation of all structures which did reference one of the stuctures $s1$ and $s2$. So $s3$, $s4$ and $s7$, which are transformed to $s3'$, $s4'$ and $s7'$, which have now a reference to the new structure $s2'$. Further structures which reference structures which have been transformed, are transformed as well, and all constraints which are connected transformed structures are transformed as well, such that possibly the operation can escalate such that all

structures and constraints of the schema are transformed. But, the group operation does not repeatedly transform a structure that has already been transformed in the same transformation step, such that the termination of the transformation operation is given.

Preceders. Structure $s2'$ gets preceder set $\{s1,s2\}$, constraint $s3'$ has preceder $s3$, $s4'$ has preceder $s4$, $s7'$ has preceder $s7$, and so forth for the structures and constraints which are additionally transformed, because they had a reference to $s2$, $s3$, $s4$, $s7$, ...

separate s1 [s2]. The separate operation separates one or more structures (in Figure 5.3 only one structure, which is contained as set-, bag-, list-, or record-typed attribute in $s1$) from a structure ($s1$). The structures, which shall be separated must be given the function by the second list-parameter. If the first structure ($s1$) is not contained in this list, it is internally added to it. If only one structure (assume $s2'$) is separated from the original structure ($s1$), the function subtracts the references from and to $s2'$ from $s1$ and generates $s1'$ and $s2'$ as new structures. The transformation of additional structures and integrity constraints and the escalation on the transformation of additional structures and integrity constraints is similar those of the group operation. Also the generated reference and cardinality constraint between the new $s1'$ and $s2'$ is similar the handling of the group operation, except that it is in reverse order.

Preceders. Both structures, $s1'$ and $s2'$, get preceder set $\{s1\}$. The preceders of the integrity constraints (cardinalities), and the additional structures and integrity constraints, which are generated by escalation of the transformation, are set corresponding the preceder settings of the group operation.

nest S1 S2 C . The nest operation (Figure 5.4) nests a structure S2 into S1. S2 is transformed to a nested attribute of S1' (here, the set-typed attribute s2'). The transformation of additional structures and integrity constraints and the escalation on the transformation of additional structures and integrity constraints is similar those of the group operation.

Preceders. S2' gets preceder set {S1,S2}.

New Attributes. Attribute s2' represents the nested structure S2.

unnest S1 a . The unnest operation separates a nested attribute (array-, set- or bag-typed, in Figure 5.5 the set-typed attribute a) from a structure (S1). The result of this operation are two new structures (S1',A'). The transformation of additional

structures and integrity constraints and the escalation on the transformation of additional structures and integrity constraints is similar those of the group operation.

Preceders. S1' gets preceder set {S1}.

New Structures. A' represents the unnested attribute a.

clusterize {S1,B,C,D} . The clusterize operation takes a relationship structure (S1), its referenced cluster and the structures which are referenced by the cluster (B,C,D), and generates one internal structure (S1'). The transformation of additional structures and integrity constraints and the escalation on the transformation of additional structures and integrity constraints is similar

Preceders. S1' gets preceder set {S1}, E' gets preceder set {B,E}, G' gets preceder set {C,G}, I' gets preceder set {D,I}.

New Structures and References. S1' does not anymore reference the cluster, instead S1' optionally references now F, H and J. S1' is now referenced by E', G' and I'.

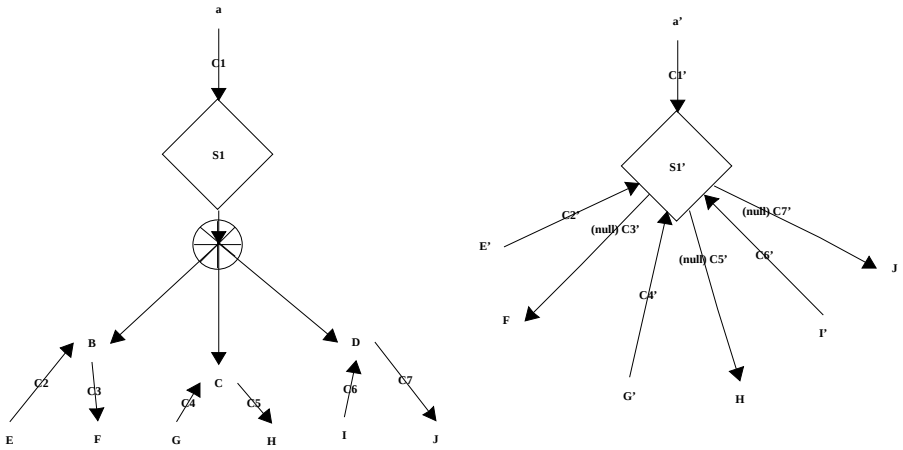


Figure 5.6: clusterize {S1,B,C,D}

There is also a transformation operation, called *mk_kettentity*, which we have not exported, and can therefore not be used in transformation rules which are specified by the database designer. The mk_kettentity transformation operation is called from the separate transformation operation, and provides the generation of *kett-entities* for a schema which has mutual references, such as the internal schema in Figure 4.11. It is used for network and object-oriented target schema generations. These data models do not allow

to have mutual references in the schema (which are many-to-many relationships on the conceptual representation), such that such references are resolved by kett-entities. Although the hierarchical data model does also not allow to have mutual references, here, in contrast to the network and object-oriented model, transformations of mutual references are omitted, since they are never generated during the transformation– for reason that the given conceptual schema is always hierarchical and there must be no hierarchical transformation rule that makes an intermediary schema non-hierarchical.

5.3 Cost Evaluation and Reflection of Internal Transactions to the Conceptual Schema

RADD/raddstar's cost model is comprised of

- *basic operation costs,* these are used to evaluate complexities of retrievals, inserts, deletes, and updates,
- *transactions extensions,* which are describing the contents of transactions that are generated by an insert, delete, or update operation, and
- *transaction graph mappings,* which translate the transactions of the internal schema which is derived from the conceptual schema to the transactions the designer is looking at (conceptual transactions), and evaluate their costs.

The basic operation costs are evaluated independently from integrity maintaining actions. The transaction extensions evaluate transaction graphs, which contain the actions and triggered actions which are necessary to generate or restore consistent database states. The basic operation cost terms are attached to the nodes of the transaction graphs and are furthermore used in equations to estimate the relative time and performance of the running database's transactions.

The functions and parameters used to evaluate the costs of the database operations are presented in Section 5.3.1. Section 5.3.2 describes the rewriting of the database operations to their *specializations.* Specializations are the sets of the operations and all operations which they depend on, whenever an operation, such as insert, delete, or update is invoked by the DBMS. Then, integrity maintenance possibly requires to invoke other operations (if cascade, set null, or set default is specified)– or to reject the invoked operation whenever the other operations on which the invoked operation is depending, were not previously executed (if restrict is specified). As mentioned at the end of Section 3.3.3.1, the specializations evaluated by the RADD/raddstar are similar to the specializations evaluated by the *GCS* algortihm, which considers the case on integrity enforcement (cascade) [ST92, ST94b]. In Section 5.3.3 we describe how the *internally evaluated* operation contents and costs are mapped to operation contents and costs of the conceptual schema, and how they are presented to the database designer.

5.3.1 Evaluation of the Basic Operation Costs

Several approaches to fitness evaluation of database operations and to query optimization in relational databases have been proposed, e.g. [Wie87, Gil91, CBC93]. The authors of reference [CBC93] use a refinement of the involved actions which are necessary to perform selects, inserts, deletes, and updates. In RADD/raddstar we use a similar refinement and evaluate the costs of the basic actions that are necessary to perform the retrieve (select), insert, delete, and update operations firstly. For the evaluation of the basic operation costs we define three types of terms:

1. *cost primitive functions*,
2. *cost parameter functions* (related to physical access and modification costs), and
3. *balancing parameters*, which are used to balance the costs of the different operation and action types upon themselves.

which are calculated by the cost parameter functions.

$p_{loc}(R^t)$	locate the data item in R^t
$p_{sto}(R^t)$	store the data item in R^t
$p_{rem}(R^t)$	remove the data item from R^t
$p_{mod}(R^t)$	modify the data item in R^t
$p_{fet}(R)$	fetch the data item of R^t
$r_{idx}(R^t)$	reorganize the related indices of R^t
$r_{buc}(R^t)$	reorganize the hash buckets of R^t
$r_{sto}(R^t)$	reorganize the whole storage of R^t

Table 5.1: Cost Primitive Functions used by the RADD/raddstar.

5.3.1.1 Cost Primitive Functions

The cost primitive functions describe the costs of actions which are necessary to maintain the data and the files in which the data are stored physically, e.g. locating the tuple, fetching the tuple, or storing the tuple. We do not consider costs of actual internal actions and disk management operations since they depend on the implementation of the DBMS and the characteristics of the physical file and directory organization. We use rather abstractions of these different physical actions. This way, the cost primitive functions state a couple of terms which are used to assemble a general framework that can be adapted by parameters according different issues. The cost primitive functions used by RADD/raddstar are shown in Table 5.1. In the table, R denotes a structure, and R^t

a)	Heap	$p_r(R^t) = n_h * log_2^{tuple\#(R^t)}$
b)	Linked Lists	$p_r(R^t) = n_l * tuple\#(R^t)$
c)	ISAM	$p_r(R^t) = n_i * (1 + log_{fanout}^{tuple\#(R^t)})$
d)	Traditional Btrees (Sparse Btrees)	$p_r(R^t) = n_b * (1 + log_{blocksize/reclen(R)}^{tuple\#(R^t)})$
e)	Dense Btrees	$p_r(R^t) = n_b * (1 + log_{blocksize/keylen(R)}^{tuple\#(R^t)})$
f)	Extensible Hashing	$p_r(R^t) = n_h' * (1+\rho_{collision}(R^t)) * \frac{n_h'' * keylen(R)}{n_h'''}$
g)	Wisconsin Storage System (WiSS)	$p_r(R^t) = n_{pw} * n_w * (1 + log_{blocksize/keylen(R)}^{tuple\#(R^t)}) * n_{pw}' * n_w' * (1+\rho_{collision}(R^t)) * \frac{n_w'' * keylen(R)}{n_w'''}$

Table 5.2: Cost Parameter Functions used by the RADD/raddstar

denotes the occurence set of R at time t– that means the set of all instances of structure R at a special point of time t –, respectively.

5.3.1.2 Cost Parameter Function

The cost parameter function relates to the type of physical data organization. We have realized cost parameter functions on the basis of Heap data organization (Heap), linked lists as used by network DBMSs (LList), index sequential access method (ISAM), sparse clustering and dense non-clustering Btrees (Btree,DBtree), extensible Hash(ing) (EHash), and wisconsin storage system (WiSS) data organization. The WiSS is a combination of dense Btrees (e) and extensible Hash (f), and chooses automatically the storage structure, that is the more appropriate. In the WiSS, also the three width of the Btrees (that is the number of children tree nodes per level, called *fanout*) and the hash bucket size (that is the maximum number of pointers of each of the boxes on the left of the picture labeled (f) in Figure 5.7, all these boxes together are called the *bucket space*) are more flexible than in the normal dense Btree (e) and Hash (f) organization. This enables the WiSS to store value sets of collection-typed and record-structured attributes. As mentioned in Section 3.3.2.3, the underlying storage manager of O_2 is based on the WiSS. (For the WiSS, refer to [CDKK85].)

Table 5.2 shows how the cost parameter function $p_r(R^t)$ is evaluated for the physical data organizations. In the table, the function $tuple\#(R^t)$ denotes the number of tuples (tuple number) of R^t. Of course, this is not the real tuple number of the concrete database

instance, since we are still in the phase of database design. The database is not realized yet, and a structure (R_1) has not already instances which are her occurence set (${R_1}^t$). But, we acquire an approximation of the tuple numbers from the database designer, and, in the case that no tuple numbers is given for some structure we use a heuristics based on the modeled integrity constraints, to approximate the tuple number of that structure.

E.g. if the database designer specifies that ${R_2}^t$ has 859 tuples, and there is a cardinality constraint $card(R_1, R_2) = (2, 5)$, then ${R_1}^t$ must have between 1718 and 4295 tuples. In this case, we assume that $tuple\#{R_1}^t$ is $\sqrt{1718 * 4295} \doteq 2716$.[1] The items $n_{...}$, $fanout$, and $blocksize$ in Table 5.2 are balancing parameters, which are described below.

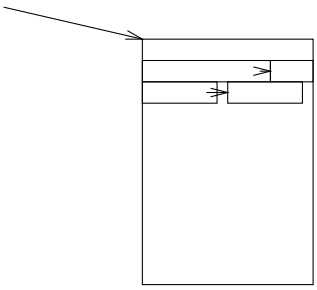
5.3.1.3 Balancing Parameters

The more specific criteria for the cost approximations of the data manipulation operations (retrieve, insert, delete, update) are provided by the balancing parameter sets. These are used for the possible combinations of transformation type (hierarchical, network, relational, ...) and storage organization (Heap, LList, ISAM, Btree, ...). The balancing parameters of the several sets are initially set to default values, and can be configured by the ".raddstar" startup file and the `<filename>.csl` CSL specification files, as well as by the graphical user interface (GUI) of the RADD/raddstar. By means of the GUI, the cost evaluation can be easily adjusted according the database designer's wishes.

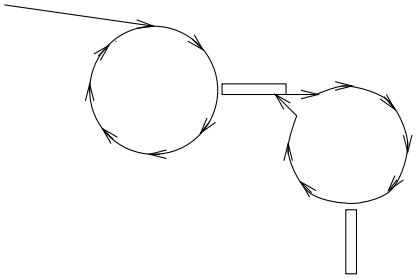
For the cost parameter functions we use the following balancing parameters:

1. Heap organized databases are generally scanned from the beginning to the end, until the searched item is found. However most DBMSs who support this type of storage organization (for instance, hierarchical DBMSs and the relational *Ingres* DBMS) perform a search which is based on a sorted binary index-tree, according a given indexed attribute or attribute set. The cost parameter function considers this frequent option of heap organized databases. We use the balancing parameter n_h here, to give the evaluated cost term the correct weight.
2. The cost parameter function for Linked Lists assumes that the average cost is in general linear to the associated tuple number (occurence set size). The record sets are scanned completely, since they are not indexed (`full table scan`). Here, we use the parameter n_l to give the costs the correct weight. (The function for Linked Lists can also be used to provide approximations for an object-identification mechanism if this is physically realized via arrays or vectors.)
3. For ISAM organized data we assume that the access costs are logarithmic with the

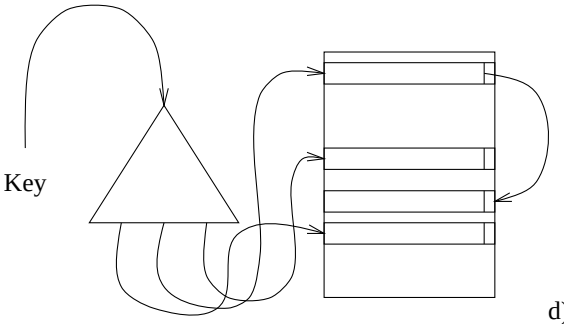
[1] The heuristics that we have implemented also works, if for none of the structures the database designer specifies a tuple number.



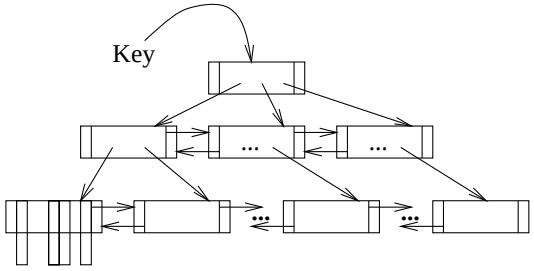
a) Heap



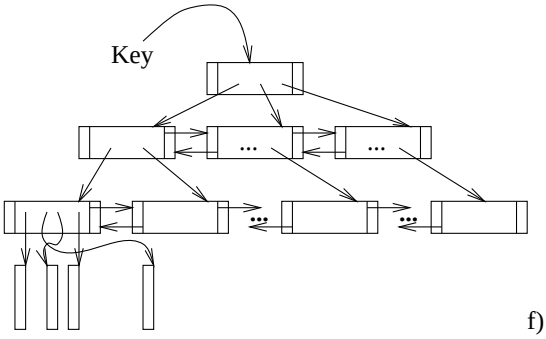
b) Linked Lists (as used by network DBMSs)



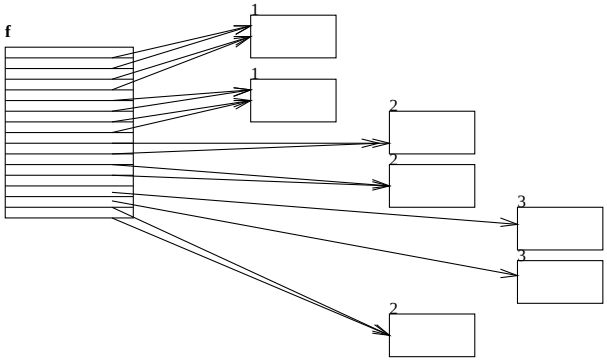
c) ISAM (Index Sequential Access Method)



d) Btree (traditional, leafs contain the indexed data records)



e) Btree (dense, leafs have pointers to the data records)



f) Hashing (location of data records evaluated by a function)

Figure 5.7: Different Kinds of Physical Data Organization.

tuple number and fanout. Here, n_i as well as $fanout$ are the parameters which influence the result of the cost parameter function.

4. Traditional Btree (sparse Btree) organized databases do data accesses that are logarithmic with the tuple number and the fraction of the blocksize (e.g. 4096) and record length.
5. Dense Btree databases perform accesses that are logarithmic with the tuple number and the fraction of the blocksize and key length. In case of tuple storage, removal, modification, fetch, and secondary storage reorganization, the cost parameter function uses the record length ($reclen$) instead of $keylen$.
6. In the case of (extensible) Hash(ing) we make special assumptions, about some parameter function $\rho_{collision}()$, which describes the probability that a bucket overflows, and, the whole bucket space must be reorganized. These situations can only occur on insertions and updates of data. We assume that $\rho_{collision}()$ is dependent upon the chosen bucket size bc, the frequency of modification requirements fm(,) and some uniformity uf() of key data according to the (maximal) hash prefix:

$$\rho_{collision}(R^t) = \frac{fm(op,R^t) * uf(R^t)}{bc}$$

   fm(,) itself depends upon the occurence set R^t and the operation type – on deletes or retrievals it is 0, and on updates it is only relevant if the updated columns are hashed. uf() depends on the tuple number $uf(R^t) = {n_h}''' * tuple\#(R^t)$, and bc is the bucket size.
7. The WiSS cost parameter function is a combination of the evaluation functions of dense Btrees and extensible Hash. Here, we use the parameters n_{pw}, n_w, ${n_{pw}}'$ and ${n_w}'$ to schedule the probabilities which physical storage mapping has been taken. The balancing parmeters ${n_w}''$ and ${n_w}'''$ are used the same way as ${n_h}''$ and ${n_h}'''$ in the Hash cost parameter function.

To balance the cost of the different actions upon the different operations of the considered data manipulation language (such as select, insert, delete, and update in SQL) we use the following parameters:

- b_{lo} – denotes the relative cost to locate the item
- b_i – relative cost to store the item
- b_d – relative cost to remove the item
- b_u – relative cost to modify the item
- b_f – relative cost to transfer the item from secondary storage to main memory
- b_{xo} – relative cost for index reorganization
- b_{bo} – relative cost for bucket space reorganization

- b_{so} – relative cost for reorganization of the primary data file

where b_{lo},b_{xo},b_{bo}, and b_{so} depend on the operation type o (retrieve, insert, delete, update).

In the cost model, the cost of an invoked operation (basic operation cost) is represented by a term $C^b(op_{st})$, such that op is the operation (retrieve, insert, delete, or update) and st is the structure to whose occurence set the operation is applied: op_{st}, such as $insert_{Department}$ or $delete_{Employee}$ denotes an operation that is executed by the DBMS.

Examples for evaluating cost terms:

- For a relational database which uses dense Btrees we assume the cost to retrieve one data record as follows:

  $C^b(retrieve_{R^t}) = p_{loc}(R^t)+p_{fet}(R) = b_{lr}*n_b*(1 + log^{tuple\#(R^t)}_{blocksize/keylen(R)})+b_f*\text{reclen}(R)$

  where $p_{loc}(R^t)$ is the time to locate the item and $p_{fet}(R)$ is the time to transfer the item to main memory.

- The operation complexities for retrievals, inserts, deletes and updates in extensible Hash organized databases are as follows:

  1. $C^b(retrieve_{R^t}) = p_{loc}(R^t) + b_f*p_{fet}(R)$
  2. $C^b(insert_{R^t}) = p_{loc}(R^t) + b_i*p_{sto}(R^t) + b_{bi}*r_{buc}(R^t) + b_{si}*r_{sto}(R^t)$
  3. $C^b(delete_{R^t}) = p_{loc}(R^t) + b_d*p_{rem}(R^t) + b_{bd}*r_{buc}(R^t)$
  4. $C^b(update_{R^t}) =$
     (a) $p_{loc}(R^t)+p_{mod}(R^t)+b_{bu}*r_{buc}(R^t)$ (non-destructive)
     (b) $p_{loc}(R^t) + p_{mod}(R^t) + b_{bu}*r_{buc}(R^t) + b_{su}*r_{sto}(R^t)$ (destructive)

Comments:

> The traditional sparse Btree cost model uses the same functions as the dense Btree model except that the *reclen* is used in any case (instead of *keylen*), and, the balancing parameters are configured differently. The Hash data organization may retain allocated memory space for data records if the data record is updated on its hashed attributes ("non-destructive"), or free this space ("destructive"). This is considered by the balancing parameter b_{su}. For the WiSS data organization type we use the dense Btree model to evaluate complexities for ordinary structures and the extensible Hash model for structures that are grouped into other structures (nested structures), and for nested (collection-typed, record-typed) attributes,

At this point, we can continue the conceptual schema analysis process by inspecting what happens on the physical schema according transactions and their dependencies.

5.3.2 Transaction Extensions

Transaction extensions are used to reprensent the sequence (or set) of operations that is invoked by a single update operation, considering the maintenance of integrity constraints (by keys or foreign-keys) and the firing of associated database triggers. The transactions are represented by (invoked) insert, delete, or update operations, and the resulting sequences (sets) of subtransactions which contain retrieve (select), insert, delete, and update operations. These sequences are enriched by control flow elements, such as *if-then-else-fi*-statements, brackets ("(" ... ")"), *null* operations ("skip"), raising of errors (whenever conditions occur, that are not appropriate), messages with which the DBMS notifies the applications about these errors, and error handling, to not abort transactions whenever errors are raised from triggered subtransactions.

In [Ste95] we have presented a rule set named *event-constraint-condition-action* (EC^2A) that contains 14 restrict, 16 cascade, and 14 set null or set default rules, with which it is possible to derive the set of subsequent operations of the invoked insert, delete, and update operations directly from the given data schema and its behavior rules. The EC^2A rules describe how a correct database on that a possibly integrity violating operation is applied is transferred to a database which again has a corrent state.

5.3.2.1 Computing the Transactions

The EC^2A rule model that is implemented in the RADD/raddstar is applicable in general, i.e., for the ability to specify structural constraints (keys and references), the approach of rule triggering systems, and the integrity maintenance by application programs. In this context, we do not care whether there is an implemented key or reference (e.g., a foreign-key and, maybe, an *on delete cascade*-clause, in a relational SQL database), or, there is a rule (trigger), database procedure or procedure in the application program, which is maintaining the integrity constraints.

The transformation operations group, nest, and clusterize (refer to Section 5.2.2), delete references and cardinality constraints from the schema. This way, the internal schema that is generated by the transformation, is simpler than the given conceptual schema because integrity constraints have been dropped. Nevertheless, a range of integrity constraints is also present in the internal schema, and, the transformation generates some new integrity constraints, which must be maintained additionally. The internal dependencies between structures (references) and the new cardinality constraints, FDs, IDs, and EDs are rewritten to operation dependencies by the RADD/raddstar.

Example. Assume there is a structure R_1 which references another structure R_2. Then, the $insert_{R_1}$ operation either demands that the R_2 object which it wants to set the reference to, does already exists in ${R_2}^t$ ("ON INSERT RESTRICT"), or otherwise $insert_{R_1}$ must trigger $insert_{R_2}$ ("ON INSERT CASCADE").

5.3.2.2 Finiteness Condition of the Transaction Content Rewriting

However, in the latter case of the examples in Section 5.3.2.1 ("ON INSERT CASCADE"), it may be that the new object in ${R_2}^t$ wants to insert again new objects into ${R_1}^t$, e.g. if there is a cardinality constraint $card(R_1, R_2) = (2, 5)$; that means each object of ${R_2}^t$ has at least 2 and at most 5 R_1 objects. Then a normal rewrite system which derives from the reference R_1 to R_2 that $insert_{R_1}$ triggers $insert_{R_2}$, and from this cardinality constraint that $insert_{R_2}$ triggers $insert_{R_1}$, will not terminate. Let us illustrate such a situation by another example. Figure 5.8 shows a database schema which is cyclic. Therefore, the operations which are implied by the schema at top of Figure 5.8 are cyclic as well. Every insert into some set (${R_1}^t$, or ${R_2}^t$) triggers a new insert into the other set. Even, if we have many structures which are participating in the cycle it seems to be a bad task to detect and to evaluate the behavior of the invoked insert operation.

Therefore, we evaluate the operations as if each operation on some occurrence set returns to the same object (and stops) once an operation of the same type (insert, delete, update) which is related to the same set is required, that is found in the set of the previously evaluated operations. This lets the transaction evaluation process terminate at any time. Such a "finite" evaluation of the operations is shown in Figure 5.9.

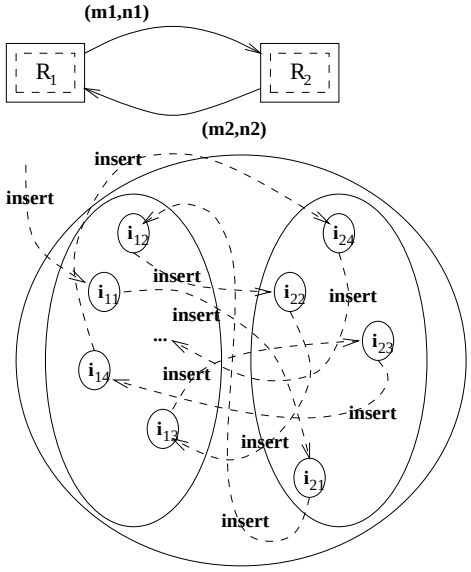


Figure 5.8: Mutual dependent Structures.

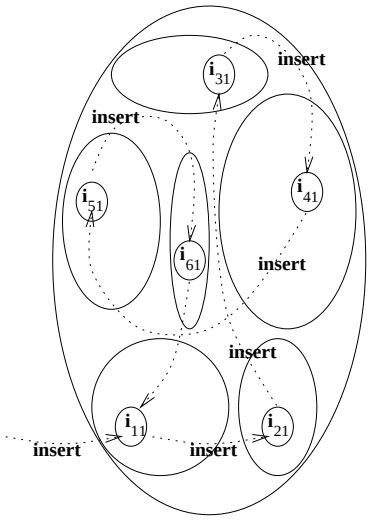


Figure 5.9: Adding the Finiteness Condition.

5.3.2.3 Behavior Computation by Rules

The insert operations in Figure 5.8 and Figure 5.9 are examples of operations sequences which are used by rule triggered integrity corrections. We presuppose that using tra-

ditional DBMSs similar situations can occur– e.g. if applications are used to maintain integrity constraints. Therefore, we can use rules to represent behavior on the internal representation (internal schema). As mentioned above, these rules are applicable in general, i.e. the ability to specify integrity enforcement by structural constraints ("ON DELETE CASCADE"), the approach of rule triggering systems, or integrity maintenance by application programs.

Correct Database States. Assume, structure R_1 has a reference to structure R_2 and there is a related cardinality constraint $card(R_1, R_2) = (m, n)$. If we restrict us to insert and delete as invoked operations, and to cascade and rollback as behavior options, and further we assume that the database state was correct, then we can describe the transitions which produce a correct database state again as follows:

1. On insert of an i_1 into ${R_1}^t$:
   - if there is no i_2 in ${R_2}^t$ such that i_1 has a reference to i_2 then insert such an i_2 into ${R_2}^t$ an check if $m > 1$: in this case invoke now $m - 1$ inserts into ${R_1}^t$ such that these new objects of ${R_1}^t$ have also a reference to i_2; otherwise, rollback the transaction.
   - if there is an i_2 in ${R_2}^t$ such that i_1 has a reference to i_2 and the number of all i_1's which have references to i_2 is already greater than or equal to m and also the number is lower than n then skip; or otherwise, the number is equal to n, either delete one object from ${R_1}^t$ which has a reference to i_2, or else, rollback the transaction.
2. On insert of an i_2 into ${R_2}^t$:
   - if $m \geq 1$ then invoke m inserts of objects into ${R_1}^t$ which have a reference to i_2; or otherwise, rollback the transaction.
3. On delete of an i_1 from ${R_1}^t$:
   - if there were m objects in ${R_1}^t$ which have references to the object in ${R_2}^t$ as i_1 has then invoke an insert of another object into ${R_1}^t$ which has now a reference to the same object in ${R_2}^t$ that i_1 had; or otherwise, rollback the transaction.
4. On delete of an i_2 from ${R_2}^t$:
   - if there were objects in ${R_1}^t$ which have references to i_2 then delete all these objects from ${R_1}^t$; or otherwise, rollback the transaction.

From these informal descriptions, we can obtain rules for integrity maintenance. Let us consider a schema with mutual dependencies where the structure R_1 has a reference to the structure R_2, and R_2 has a reference to R_1 such that $card(R_1, R_2) = (m1, n1)$. The arrow from R_1 to R_2 specifies implicitely a $card(R_2, R_1) = (1, 1)$, i.e. every object of ${R_1}^t$ has exactly one object in ${R_2}^t$. Together with the cardinality constraint $card(R_1, R_2) = (m1, n1)$, this can be used to derive the following rules:

1. ON INSERT$_{R_1}(i_1)$:
   if $\nexists i_2 \in {R_2}^t : i_1 \rightarrow i_2$ then INSERT$_{R_2}({i_2}'|_{i_1 \rightarrow {i_2}'})$
   else
   if $|{i_1}'|_{{i_1}' \rightarrow i_2 \in {R_2}^t}| = n$ then ROLLBACK else SKIP fi
   fi
2. ON INSERT$_{R_2}(i_2)$:
   m1 * INSERT$_{R_1}(i_1|_{i_1 \rightarrow i_2})$
3. ON DELETE$_{R_1}(i_1)$:
   if $|{i_1}' \in {R_1}^t|_{{i_1}' \rightarrow (i_1 \rightarrow i_2)}| \leq m$ then ROLLBACK else SKIP fi
4. ON DELETE$_{R_2}(i)$:
   $|i_1 \in {R_1}^t|_{i_1 \rightarrow i_2}|$ * DELETE$_{R_1}({i_1}' \in {R_1}^t|_{{i_1}' \rightarrow i_2})$

Remarks. Firstly, in this rule set it doesn't matter whether R_2 has also a reference to R_1, as the schema in Figure 5.8 specifies. This situation (of Figure 5.8) is considered by another rule set that must be combined with the above rules. Secondly, we have omitted operations which are probably producing undesired results (e.g., an insert operation that triggers a delete operation). But, if we have a general constraint set, such as FDs, IDs, and EDs together, then delete operations can also appear in the operation sequence of an insert operation, or insert operations may also appear in the operation sequence of a delete operation, although the delete operation may not be triggered immediately by the insert, and the insert operation may not be triggered immediately by the delete.

5.3.2.4 Estimating how often Operations require other Operations

In the rule set we show how often integrity repairing actions are necessary, e.g. by the m1 * INSERT ... . From these numbers (m1) we can derive *plausibilities* which form a weight describing the probability we expect a database operation to trigger or require another database operation.

The plausibility Function. Let o_1, o_2 be update operations, $o_1, o_2 \in$ {insert, delete, update}. R_1, R_2 are structures and the parameter types of o_1, o_2. R_1 has a reference to R_1, e.g. R_1 is a relationship structure and R_2 is an entity structure to which R_1 has an arrow. Let the associated cardinality constraint be $card(R_1, R_2) = (m, n)$. Then, to estimate how often an operation on set ${R_2}^t$ triggers or requires– such that the transaction may be rolled back –an operation on set ${R_1}^t$, we evaluate the following plausibility:

$$\beta_{o_2 R_2 o_1 R_1} = \frac{avg(m,n)*\sqrt{m}}{\sqrt{n}} * \frac{\sqrt{tuple\#(R_2)}}{\sqrt{tuple\#(R_2)}+\sqrt{tuple\#(R_1)}}$$

Otherwise, to estimate how often an operation on set ${R_1}^t$ triggers or requires an operation on set ${R_2}^t$, we evaluate:

$$\beta_{o_1 R_1 o_2 R_2} = \frac{\sqrt{n+1}}{avg(m,n)*\sqrt{m+1}} * \frac{\sqrt{tuple\#(R_1)}}{\sqrt{tuple\#(R_1)}+\sqrt{tuple\#(R_2)}}$$

For reason that update operations must refer to both cases each time (required by insert and delete), here we do something special. In general, we assume the necessity for required suboperations 0.5 times as much as the necessity for the corresponding insert's or delete's suboperations. But, the user can change the multiplier according the DBMS's transaction processing. E.g., if the expected DBMS performs destructive updates (updates are realized by deletes and subsequent inserts), the multiplier can be set to 1. This effects that the complexity of the update operation is evaluated by considering the completely weighted complexities of the according insert's and delete's suboperation sequences.

The plau_card Function. The cardinality constraint which is required for the cost estimation may be not specified or may be specified only partially. We set $m = 0$ if the lower bound is not given. If the upper bound is unspecified then we use $n{=}max(m * 3, 20)$. For the computation of the plausibilities, the related cardinalities of functional dependencies are treated like (1,1)-cardinality constraints, and those of inclusion and exclusion dependencies are treated like (1,.)-cardinality constraints. These cardinalities are evaluated by the *plau_card* function, which also determines cardinalities on arrows (references) for which no cardinality constraints are specified. The plau_card function considers the whole database schema. That is, the set of all structures, the set of all integrity constraints and the tuple numbers which either are specified by the database designer or else are inferred by the heuristics: From other tuple numbers, the structure references, and the associated integrity constraints.

The plau_card function is also applied whenever conceptual structs and unions are transformed to record, list, set, or bag typed attributes of the internal schema, or attributes of the conceptual schema are represented by internal structs or unions.

For integrity maintaining operations on structures that are transformed to internal set structures, we do something special. Assume R_1 has a reference to R_2, and R_1 has also a reference to R_3. The transformation groups R_2 and R_1 to some new structure R_2' with a set-typed attribute of R_1. Hence, the new structure R_2' has references to R_3 in the internal schema. Then the plausibility between $R_2{}'$ and R_3 ($\beta_{o_2{}_{R_2'}\,o_2{}_{R_3}}$) is multiplied by the term:

$$\frac{tuple\#(R_1)}{tuple\#(R_2)}.$$

5.3.3 Transaction Graph Mappings and Cost Evaluation

For the computation of the transaction costs, the RADD/raddstar infers transaction graphs from the internal schema and the conceptual schema. They represent the invoked operation and annotated operation sequences, which are together necessary to perform the insert, delete, or update operation. Retrieve (or select) operations are not mapped

to real transaction graphs since they do not require other suboperations, i.e. retrievals, inserts, deletes, or updates. The inference process is performed continuously until an equal operation call (insert, delete, update) on the same set is detected (for reason of the finiteness condition that we apply, refer to Section 5.3.2.2). Then, the algorithm stops for termination reasons. By considering the subtransaction sequences, inconsistences can be detected – e.g., an insert operation that undesirably triggers a delete operation from the same set. If such a situation is recognized, the RADD/raddstar notifies the database designer that he probably specified an inconsistent conceptual design.

RADD/raddstar uses the transaction graphs to construct equation systems. The root node which represents the invoked operation and its first level children nodes are mapped to a cost equation. Let us assume that, in a *Company, Employees are working on Projects,* and the works_on structure is repesented by a nested attribute of the Employee structure of the internal schema.[2] Consider the left-most double-lined node labeled Insert(Emp) in Figure 5.10, where Emp and Proj represent the structures of the internal schema. The complexity of the $insert_{Emp}$ operation is evaluated by the equation

$$C(insert_{Emp}) = C^b(insert_{Emp}) + C(retrieve_{Proj}) + \beta_{insert_{Emp}insert_{Proj}} * C(insert_{Proj}).$$

The set of all cost equations of the internal schema is then solved, and the costs for the according nodes are annotated on the transactions graphs.

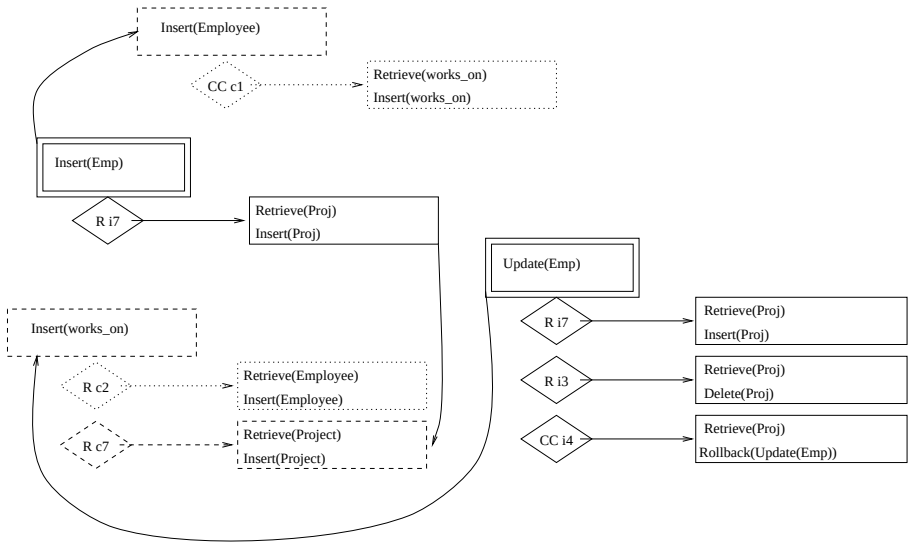


Figure 5.10: Mapping Transaction Graphs to evaluate Conceptual Transaction Costs.

[2]Using Oracle8 [Oracle8] this can be implemented by a works_on attribute with type varray[n] of integer, such that each element of the array is a foreign-key to the Project relation, for example.

Transaction graphs are evaluated for the conceptual schema too, such that the contents and the costs of the internal schema transaction nodes are mapped to transactions that the designer is looking at. The arrows in Figure 5.10 show how the costs of the transaction nodes of the internal schema are mapped to the transaction nodes of the conceptual schema. Here, the double-lined nodes contain the basic operation costs of the internal transactions, and the subtransactions of the internal transactions are represented by the nodes with the solid lines. The subtransactions relate to the causing constraint of the internal schema, respectively. The dashed nodes and lines show the transactions which are presented to the database designer (conceptual transactions). The dotted nodes which have no equivalent node in the sequences of the internal schema transactions are omitted. This way, the internal transactions' contents are used to evaluate the conceptual transactions' contents.

The conceptual transactions' cost evaluation. The conceptual transactions' costs are not evaluated by equations for the conceptual transactions and their subtransactions, but by their equivalents on the internal transactions. To enlighten this, let us consider the following example: The transaction contents of $insert_{Emp}$ and $update_{Emp}$ are mapped to conceptual $insert_{Employee}$ and $insert_{works\_on}$ operations, such that the conceptual transaction contents are based on the following internal actions

- $insert_{Emp}$ which rewrites to $insert_{Emp}, retrieve_{Proj}, insert_{Proj}$, and
- $update_{Emp}$ which rewrites to $update_{Emp}$.[3]

Thus, the internal transaction contents as well as their annotated costs are evaluated and presented to the database designer as

- $insert_{Employee}$ which is presented as $insert_{Employee}$, and
- $insert_{works\_on}$ which is presented as $insert_{works\_on}, retrieve_{Project}, insert_{Project}$.

Finally, a heuristics performs a cost value adaption such that the database designer does not wonder about the composition of the conceptual transaction cost value. I.e., the conceptual transaction cost is made the sum of the transaction's basic operation cost and the costs of its subtransactions.

[3] And, some other operations which are not important for the evaluations considered here.

5.4 Summary and Outlook

The Chapter shows how transactions and transaction costs of the internal schema are evaluated by the RADD/raddstar. The internal transactions' contents and costs of the several nodes of the generated transaction graphs are then mapped to "conceptual" transactions, which are presented to the database designer. Since the framework for the transaction cost evaluation is originated by the author, and was presented in [Ste96], we have set only a few external references in this Chapter.

For detailed physical aspects such as the behavior and time consume of read(), write(), or lseek() disk operations and hard-disk management, the interested reader is directed to [Wie87] and [KS91]. In [CBC93] an approach was presented to compute optimal indices for relational databases. This approach uses basic operations and therefore, complexities for (sub-)operations that are necessary to perform selects, inserts, deletes, and updates in a relational database– similar to our evaluation of basic operation costs. For the definition of transaction extensions, and how they are used to represent operations which are invoked from (other) insert, delete, or update operations refer to [CFPT94, RR94]. If we have a specification of a more general constraint set, such as functional, inclusion, and exclusion dependencies together, then delete operations can be triggered by a rule for an insert operation, although the deletes are not triggered directly by the insert. We do not consider this in special details in the RADD/raddstar, but we notify the database designer on the probably inconsistent schema specification. The evaluation of transaction specializations in the RADD/raddstar is related to the *greatest consistent specialization* (GCS) approach of *Schewe and Thalheim* [ST92]. For exhibitions on this approach, the interested reader may refer to [ST94a, ST94b, SST94, ST98].

The transaction and transaction cost evaluation, the *plausibility* and *plau_card* function, and the transaction cost mapping, as presented in this Chapter, are used in the RADD/raddstar to identify bottlenecks of the conceptual schema, and to gather criteria for better schema design and automatic conceptual schema restructuring. We will refer to these concepts in Chapter 8.

Chapter 6

Type Inference and Functional Schema Representation

The construction of a database design tool such that

1. the basic constructs of the data schema are considered,
2. additional requirements can be introduced to the schema, and
3. properties can be derived from it, in order to improve the schema,

needs a formal theoretical foundation ([Bac91]). This is the more important, since we provide transformations of the database schema, and store the transformations that were applied to the schema *(the schema transformation history)* by the schema itself.

Since some first steps in order to implement the database specification and analysis tool were made using an *algebraic specification language*, and, the final tool has been realized using a functional programming system, this Chapter gives firstly an overview on algebraic specification and functional programming techniques, and defines then the RADD/raddstar data model (RADD*).

The Chapter is organized as follows. Section 6.1 gives a short example on how a database system can be described using algebraic specification techniques. Section 6.2 gives an introduction to the functional implementation and the type inference concepts, which are used for evaluation of the database designer's specifications as well as for RADD/raddstar's schema transformaion operations. Then, in Section 6.3 we define the RADD* data model, which was implemented using the Standard ML of New-Jersey functional programming language, and stores the database designer's behavior specifications of method definitions that are formulated in language close to Standard ML. Finally, Section 6.4 summarizes this Chapter and gives an outlook how the RADD* data model is used to store the specifications of the conceptual specification language (CSL), that we will present in Chapter 7.

6.1 Specifying and Analysing Databases using Algebraic Specification Techniques

Algebraic specification is used to give formal mathematical foundation and specify for the properties of objects. Abstract data types (ADTs) which are usually called classes, can be prototypically designed, or specified and analysed using algebraic specification techniques. A lot of attention and effort has once been invested in algebraic specification of database systems. The works of [EW78, EKW79, Bro87, FSS88], for example, consider a concrete discourse and specify the operations that insert the items into– delete the items from–the sets of the database. However, these works are based on a given environment, such that new aspects require a new algebraic specification, respectively. Also, the work [Sch91] which claims to give a general technique for automatic translation of a HERM database schema into an algebraic specification, refers to a concrete environment for which the according algebraic specification is given. Database specification approaches which claim to be more general, e.g. [AE91, Gog93, PCO95], on the other hand, do not consider what is really happening when a database is running under use of a special DBMS or one of a DBMS class, like a relational, object-relational, or object-oriented DBMS.

In this work, we have developed an approach which is more general than the mentioned database specification approaches. Our approach considers the generic operations that are provided by DBMSs (select, insert, delete, update), and beyond this, a database type system for that allows

1. to map all items which are identified in the underling conceptual database schema of the graphical RADD database design editor, and
2. to characterize the type system used for the conceptual schema's implementation.

Algebraic Specifications. Algebraic specifications comprise algebraic specifications that were previously designed, sorts ("SORTS") which describe the types that are used, operators ("OPS") which describe the signatures of the used functions, exceptions ("EXS") which are used to raise errors on exceptional situations, and equations ("EQS") which describe the properties of the operators by means of the state after the operator's application. For the description of the equations, variables ("VARS") are used. The variables are of the sorts which are inherited from the previously designed algebraic specifications, of the sorts which are given as sort parameters– in case of parameterized ADTs (PADTs) –, or of the sorts which are explicitely listed within the current algebraic specification. For these latter kinds of sorts, the current algebraic specification defines then the *constructors* and the *destructors* (or *selectors*).

Assume we want to describe how a structure is used and maintained by a relational DBMS. Hence, we could specify a PADT *relation* which takes the attributes *attrs* and

the key attributes *keys* as parameters. The algebraic specification defining this PADT is shown in Figure 6.1.

```
SPEC relation(attrs,keys) IS
    bool + int + tuple(attrs) + set(tuple) +
    SORTS         relation
    OPS           select : set * ( tuple -> bool ) -> set
                  project : set * attrs -> set
                  insert : set * tuple -> set
                  delete : set * ( tuple -> bool ) -> set
                  count : set -> int
    EXS           duplicate
    VARS          a: attrs; p: tuple -> bool; s: set; t: tuple;
    EQS           select (add(s,t),p) =
                   if p(t) == true then add(select(s,p),t) else select(s,p) fi
                  select (empty,p) = empty
                  project (add(s,t),a) = add(project(s,a),t.a)
                  project (empty,a) = empty
                  insert (s,t) =
                   if t.keys in project(s,keys) then raise duplicate else add(s,t) fi
                  delete (add(s,t),p) =
                   if p(t) then delete(s,p) else add(delete(s,p),t) fi
                  delete (empty,p) = empty
                  count (add(s,t)) = 1 + count(s)
                  count (empty) = 0
END SPEC
```

Figure 6.1: Algebraic Specification of a "relation" Class– with Generic "select", "insert", and "delete" Operations.

In the specifciation of Figure 6.1, the operators *insert* and *delete* may be considered as the constructors of the relation sort, and the operators *select*, *project*, and *count* may be considered as the destructors of the relation sort.

The object specification system *OBJ3* [GWM+91] which is built on top of the Lisp functional programming language, has been used firstly in this work in order to implement a type system for the RADD data model, and to prototype the specified RADD data schemata by defining problem specific database functions. For instance, we did implement SQL-like insert and delete operations which toke the structure (relation type) and the tuple to be inserted or deleted as parameters– like shown in Figure 6.1. However, the evaluation of the OBJ3 specification has been shown to be so much as complex, that the underlying lisp system could not manage the evaluation complexity when using database structures with three ore more attributes.

Therefore, we broke the experiments using an algebraic specification system as implementation language, and continued with implementing the basic data types, operations, and modules using the ML functional language. We implemented a type system for the RADD data model in *Caml-light* [Mau93] firstly, and continued then realizing schema evaluation and valuation functions using Standard ML of New-Jersey.

6.2 Functional Implementation of the RADD/radd-star

A programming language is called "functional" whenever its basic construct of program structuring is *function* and its primary control structure is that of *function application*. For example, the *Lisp* programming language [Mac62] is called a functional language because it possesses these properties. New-generation functional programming languages are not anymore as strict functional as the Lisp language, but include also declarative elements. The main advantage of new-generation functional programming languages– in comparison to procedural programming languages, like Pascal or C –is to express the return value of a function as a formal specification. Also, many functional languages provide an orthogonal use of control and user-defined constructs, such that, for example, an *if-then-else* construct can be considered as a function that returns the *then*-value if the predicate evaluates to true, or otherwise the *else*-value. A function itself can also have function parameters– then, the function is called a *higher-order function* –or in turn, it may return a function as result.

The following can be considered as new-generation functional languages: *SASL* [Tur76], *Miranda* [Tur85], and *ML* ("meta language") [GMM+78, Mil87]. SASL and Miranda perform *lazy evaluation*, that is, arguments which are passed to a function are not evaluated when the function is called, but only when they are needed within the called function. Most ML dialects do *eager evaluation*, which means that arguments are evaluated before they are passed to a function.[1]

The specification character of the new-generation functional languages provides the benefit that they are well usable for the implementation of term-rewriting systems and theorem provers, or tools for algebraic specification. Therefore, these tools are often built on top of functional languages, because the functional language has direct support for the construction of the elements which are used in the clauses and equations.

6.2.1 The Standard ML of New-Jersey Programming Language

Lisp as well as Miranda have an untyped semantics in sense that functions may be passed parameters of several data types, like integer and float. An advantage of untypedness is that a function describing the same behavior on different types, like $sqrt(n)$, can be applied to $n{:}integer$ or $n{:}float$, and is everytime preserving the same property, namly $sqrt(n)*sqrt(n) = n$. In contrast, ML has a strongly-typed semantics, that means that functions for floats can not implicitely be used for integers as well, and also *infix* operators,

[1] There exist also ML dialects which use lazy evaluation, e.g. *Lazy ML* developed at the University of Göteborg, Sweden.

like $+, -, *, /$ can not be used with arguments of different types on the left-hand and on the right-hand side.

6.2.1.1 Some Specifics of the Standard ML of New-Jersey System

Although strongly typed, polymorphic function applications can be simulated in Standard ML of New-Jersey (SML) using function or operator overloading– for which an example is shown in Figure 6.2.

```
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- Integer.+;
val it = <primop> : int * int -> int
- Real.+;
val it = <primop> : real * real -> real
- overload newplus : 'a * 'a -> 'a as Integer.+ and Real.+;
overload
- newplus;
std_in:3.1-3.7 Error: overloaded variable cannot be resolved: newplus
- newplus(1,3);
val it = 4 : int
- newplus (1.0,3.0);
val it = 4.0 : real
- infix newplus;
- 1.0 newplus 3.0;
val it = 4.0 : real
- 1.0 newplus 3;
std_in:7.1-7.13 Error: operator and operand don't agree (tycon mismatch)
  operator domain: real * real
  operand:         real * int
  in expression:
    newplus : overloaded (1.0,3)
-
```

Figure 6.2: Overloading an Operator in Standard ML of New-Jersey (SML).

Operator Overloading. In Figure 6.2, the newplus operator is overloaded with the + for integer (Integer.+) and the + for float (real, Real.+). In the signatur description $'a * 'a \rightarrow 'a$ the $'a$ is a type parameter, respectively. The signature expresses that newplus takes two arguments of the same type and returns a value of that type. Newplus was then testet on a few examples ((1,3), (1.0,3.0), (1.0,3.0), (1.0,3)) where the last two examples use newplus as infix operator. Newplus on (1.0,3) failed since the first and the second argument were not of the same type. This is for reason that given an expression like $x + y$, SML derives the most general type of the arguments, which was (int,int), (real,real), (real,real), (real,int) in the four examples, respectively. This generated the exception on the last example, which evaluated to the type combination (real,int) that does not match $'a * 'a$.

So, $1.0 + 3$ is an invalid SML expression. This enforces, that *coercion functions*, that are functions that convert the type of an argument or function parameter, must be used in

these cases. E.g. *real* is the coercion function to convert an integer to the corresponding float ('float' is sometimes called 'real', but I like the type denotation 'real', since it is also the name of the SQL type). Therefore, $1.0 + (real\ 3)$ is a type-correct SML expression.

In the example of Figure 6.2 $'a$ is a type parameter, but we could also define the newplus function as shown in Figure 6.3, such that newplus is restricted to take *equality type* parameters and also delivers a value of an equality type as result. Using an equality type is a more restrictive declaration but makes sense, since the plus operation is only known for equality types (such as int and real).

The distinction between equality types and types is that equality types cannot be function types. Reference values which are constructed using the ref function ($ref : 'a \rightarrow 'a\ ref$) have equality types as well: $'a\ ref$ is always an equality type, although $'a$ may be not an equality type.

The following signature, $''a * ''a \rightarrow ''a$, describes newplus such that it can be applied only to values of an equality type.

```
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- overload newplus : ''a * ''a -> ''a as Integer.+ and Real.+;
overload
-
```

Figure 6.3: Another Kind of Operator Overloading using SML.

Parametric Polymorphism and Union Types. Although strongly typed, SML supports polymorphism by means of *parametric types* and *union types*. Parametric types are types that have a so-called type parameter, e.g. $'a$ of the built-in parametric type $'a\ list$. Union types, on the other hand, possess an indicator for the actual type, e.g. $datatype\ 'a\ option = NONE \mid SOME\ of\ 'a$ declares type *option* as either $NONE$ (i.e. nothing), or something, $SOME$, of type $'a$.

Call-by-Value and Call-by-Reference Evaluation. As mentioned above, the evaluation regime of SML is eager evaluation. The evaluation is also strict *call-by-value*, i.e. the argument retains the same value even after the called function has terminated. Interpretative programming languages, like Basic, offer call-by-name evaluation which allows to refer to and update variables in global pools. The counterpart of call-by-value passing in procedural programming languages is *call-by-reference*. Call-by-reference argument passing can be simulated using certain ML hybrids, like *Caml Light* [Mau93] where it is possible to define *mutable* fields in record-types that can be changed by assigning them new values. For change of values (by functions etc.), SML supports *references* whose content can be newly assigned. Allowing only a reference's content to be updated enforces a clean programming style and avoids side-effects, which may lead to inconsistences and bugs.

Also, in contrast to other ML dialects, e.g. *Caml Light*, the `ref` function of SML creates a handle to a unique object which is equal to the object the ref function is applied to. This way, only the content of that reference is changed when the reference is newly assigned, but the original object remains unchanged. So, SML expressions like `ref 5 = ref 5` deliver false.[2]

SML as Implementation Language for Algebraic Specification Tools, Theorem Provers, and Database Research Tools. The theorem prover *Isabelle* has been build on top of SML at the University of Edinburgh [Pau90]. Isabelle, more specific its release of 1990, was once considered to be involved in this work, since it states a generic theorem prover supporting several different logics, and so, several different type systems. In this way, also a general type system for databases could be defined (implemented) on top of it. In addition, the *CRML* system developed on basis of SML at the Oregon Graduate Institute for Science & Technology, Portland, has been considered for this research. CRML enables to reflect structures internally compiled into an abstract machine code by the SML interpreter, to their external representation, i.e. to the SML input statements that have been used to create the internal code representation. CRML is used in the German joint research project *CROQUE* to examine and develope new query optimization strategies for object-oriented database management systems.

6.2.1.2 The SML Module System: Modules (Structures) and Parameterized Modules (Functors)

As shown by Figure 6.2 and 6.3, SML maintains modules which define data types, values, and operators, such as *Integer* and *Real*. Modules are called *structure* in SML, and have *signatures* which can be explicitely specified by the programmer, such that only the types and operators declared in the signature are exported from the structure. Furthermore, in SML not only data types can be parameterized, but also structures– like *templates* in C++ and other object-oriented programming languages. These parameterized structures are called *functor*.[3]

In SML, functors may be parameterized by one or more structures, such that the functor parameters can be restricted to previously specified signatures. E.g.,

```
functor RuleInterpreterEnv (structure Io : RULEINTPARSERIO ) : RULEINTERPRETERENV =
 struct
  ...
 end
```

[2]The ref function of SML works similar the Object-ID (*oid*) generation mechanisms in object-oriented databases.

[3]In category theory, a functor describes the homomorphical mapping from objects and arrows of one category into another.

declares the functor RuleInterpreterEnv to have a parameter structure that matches the signature RULEINTPARSERIO, and it restricts RuleInterpreterEnv to match signature RULEINTERPRETERENV. Then, if a structure (assume CIO) matches RULEINTPARSERIO, a new structure RiEnv can be defined by

```
structure RiEnv = RuleInterpreterEnv(structure Io = CIO)
```

Higher-order functors. The structures which are constructed by means of a functor can be used in turn to parameterize other functors. E.g.,

```
functor RuleInterpreterFun ( structure RE : RULEINTERPRETERENV ) : sig ... end =
 struct
  ...
 end
structure RuleInterpreter = RuleInterpreterFun(structure RE = RiEnv)
```

declares RuleInterpreterFun as functor whose parameter structures (of signature RULEINTERPRETERENV) are built by means of a functor as well, and then the actual structure RuleInterpreter is defined by using the RiEnv that was previously constructed by means of a functor.

Value Functors. Normally, in SML a functor is parameterized by a structure such that the data types, values, and operators of the parameter structure give the functor its proper behavior. However, an undocumented feature of SML is that a functor may also be parameterized by a value, function, or combination of them only. E.g.,

```
functor ExportRS ( val pr : string -> unit and perr : string -> unit ) =
 struct
  ...
 end
```

is such an application of a value parameterized functor definition.[4] The meaning of the functions pr and perr which are the parameter functions of ExportRS, is that they print a string to std_out and std_err, respectively.

6.2.2 Type Inference in Functional Languages

Procedural programming languages often have a type system which is static such that type errors are detected at compile-time. E.g., Pascal is such a language that requires the programmer to declare each variable with its type and rejects typing errors immediately. On the other hand, new object-oriented languages, like C++, offer polymorphism and operator overloading, such that the type-compatibility can not completely checked at compile-time and type errors can occur at run-time.

[4] To implement the RADD/raddstar, we once have exploited this feature to extend the **Concurrent ML** system (CML) [Rep91, Rep93] such that it simulates a multiprocessor engine.

As we have seen above, SML supports operator overloading, but it supports typed (parametric) polymorphism too:

```
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- fold;
val it = fn : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b
- fold (fn (a,b) => a::b) ["What","is","your","name","?"];
val it = fn : string list -> string list
- it["==>>","my","name","is","Martin","Steeg"];
val it =
  ["What","is","your","name","?","==>>","my","name","is","Martin","Steeg"]
  : string list
-
```

In this example, the `fold` function takes a function that converts a tuple of type `'a * 'b` to a value of type `'b`, a list of type `'a list`, and a value of type `'b`, such that finally again a value of type `'b` is returned. As the example shows, SML reduces the type parameters to their most concrete types, such that the type inference reduces `'a` to `string` and `'b` to `string list`.
Other frequently used type parametric functions of SML are

- `map : ('a -> 'b) -> 'a list -> 'b list`
- `hd : 'a list -> 'a` (head)
- `tl : 'a list -> 'a list` (tail)
- or the $\circ$-function which combines two functions (f,g) to one function ($f \circ g$)
  `o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b`.

In RADD/raddstar, we have implemented a type inference mechanism, which is based on λ-calculus and extends the previously presented *type systems and type inference systems* (see [Bru62, Mil78, Fai85, BO96]) to combine databases' and programming environments' type systems. This type system will be presented in Section 6.3.

To give the reader an introduction to the type inference techniques that are implemented in the RADD/raddstar, we will now develope a small functional language and its type system. The language supports function parameters which are passed and processed using λ-abstraction techinque as well as pattern matching. The type inference system of the RADD/raddstar specification language (CSL) and the RADD* data model (Section 6.3) are an extension of the type system and the type inference system that we present in the following.[5]

6.2.2.1 A Functional Language with λ-Abstraction and Pattern Matching

Most procedural and functional programming languages are *bootstrapable*, that is, they can be used to implement the language by the language herself. E.g., it is possible to

[5]The functional language presented here is a modified version of the Caml-light implementation of *a simple language* (ASL) that is described in [Mau93] Chapter 12 - 16, implemented in SML.

write a Pascal or C program that implements again the Pascal- or C-compiler. Although SML is rather used as an interpreter with a top-level loop that offers the programming interface to the user and evaluates the commands of the user immediately, similar features are applicable as well.

The language that we want to provide the designer to specify the database should be capable of both:

1. to use parameters which are specialized by the function body that evaluates the result, and
2. to use patterns which may derive the function signature and result immediately.

Suppose we want to define a function which evaluates the faculty of a number, then the function could be defined by the following declaration:

```
fun fac n = if n = 0 then 1 else n * fac(n-1) fi
```

or, alternatively by:

```
fun fac 0 = 1 | n = n * fac(n - 1)
```

The upper definition of the faculty function is according to *lambda*-abstraction, which means to declare the variables of the function as

```
fac = λn.(if n = 0 then 1 else n * fac(n-1) fi)
```

such that the `λn` is used to introduce the variable `n` and can be read as *for all n ....*

The lower definition of the faculty function additionally uses *pattern-matching* in the first clause (`0 = 1`), to ascertain whether the argument passed to the fac function is 0 or not. To represent these definitions internally, we must map them by means of an abtract syntax tree that contains the elements of this declaration. The syntax tree is generated by the parser of the functional language.

Parsing. Assume the syntax of the language is given by the following BNF grammar:

```
stmt   ::= <fundef> | <valdef> | <value>

fundef ::= 'fun' <ident> <parm> '=' <value> { '|' <parm> '=' <value> }*

valdef ::= 'val' <ident> '=' <value>

parm   ::= <ident>
         | <bool> | <int> | <string>
         | '(' <parm> { ',' <parm> }* ')'

value  ::= <ident>
         | <bool> | <int> | <string>
         | '(' <value> { ',' <value> }* ')'
         | 'fn' <parm> '=>' <value> { '|' <parm> '=>' <value> }*
         | 'if' <value> 'then' <value> 'else' <value> 'fi'
         | <value> <op> <value>
         | <value> <value>
```

such that <ident> are identifiers consisting of characters and numbers and beginning with a character, <bool>, <int>, and <string> are values of the according type– identifiers and these values are already recognized by the lexical analyser, and <op> are operators ("=","<","<=",">",">=","<>","+","-","*","/") which are used infix. The last value rule, "value ::= ... <value> <value>", considers applications of functions to values. The operators "=","<","<=",">",">=", and "<>" have the signature

```
'a * 'a -> bool
```

and, the operators "+","-","*", and "/" have the signature[6]

```
int * int -> int.
```

Preparation of the Compilation Process. After parsing we have to translate the included identifiers to variables and the "<value> <op> <value>" expressions to applications.

The first step after successful parsing is to initialize a variable environment which contains all identifiers that are bound to values, such as "+" which is bound to a function of type `int * int -> int`. When we recognize an identifier on the left-hand-side of a statement, that is an "<ident>" in the "fundef" or "valdef" rule or an "<ident>" which is included in the "<parm>" list of a "fundef" or in the "<parm>" after the symbol 'fn', then we transform it to a variable which is appended at the end of that variable environment. Whenever we recognize an identifier on the right-hand-side then we check the variable environment for the last occurence of a variable with that name, and if found we equalize the right-hand-side identifier to that variable. Otherwise ("not found"), we raise an exception (`SUnbound of string`) according that unbound identifier.

Expressions with infix operators are stored by `VALLIST` expressions, after parsing. Since the infix operators are similar the SML infix operators, we transform `5+7` (which is represented by `VALLIST[VI 5,VOP"+",VI 7]` firstly) to expressions that are as if we had read `+(5,7)`– that is an application of the "+"-function to the tuple `(5,7)` –stored by the term `VAPP(<+fn>,VPARMS[VI 5,VI 7])`.

Abstract Systax Trees. Assume we have coded and the operators "+", "-", "*", "/" as VVAR 0, VVAR 1, VVAR 2, VVAR 3, respectively. and the equality-operator "=" and the comparison operators "<","<=",">",">=","<>" as VVAR 4, VVAR 5, VVAR 6, VVAR 7, VVAR 8, VVAR 9, respectively. Furthermore, we consider integers (VI), tuple parameters (VPARMS), function applications (VAPP), function definitions (VFNDEF), and if-then-else-fi (VBRANCH) as node types. As mentioned above, besides the operators $=, +, -, *, /$, the new identifiers introduced by the function definition are considered as variables as well, such that fac is represented as VVAR 10 and its parameter n is represented as VVAR 11.

[6] In SML and in the RADD/raddstar specification language (CSL), the operators "+","-","*", and "/" are applicable to integers and floats (reals) such that they have the signature 'a * 'a -¿ 'a, respectively.

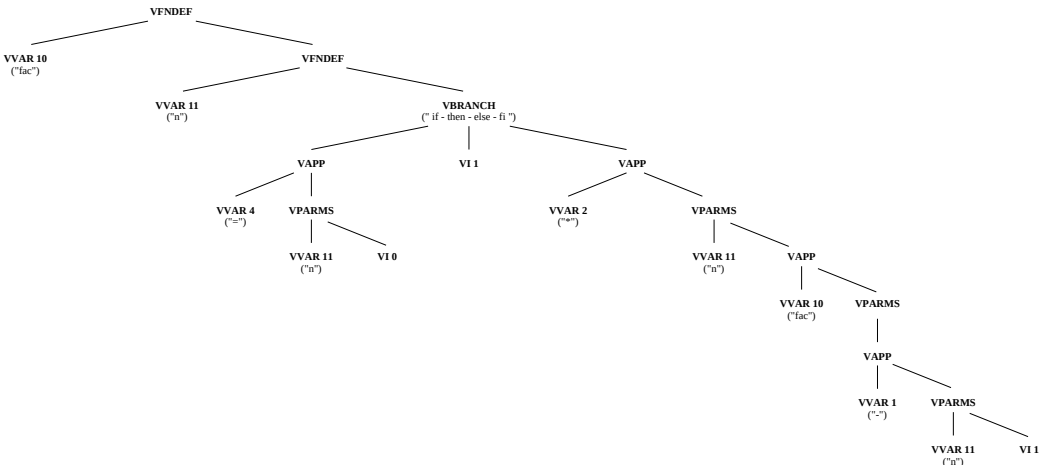


Figure 6.4: Abstract Systax Tree of the "fac" Function Definition with λ-Abstraction.

Then, the faculty function definition which was given firstly can be represented by the abstract syntax tree shown in Figure 6.4.

6.2.2.2 Representation of Basic Types and Typing Rules.

Expressions of the functional language that we present in this Section as well as of the RADD/raddstar conceptual specification language (CSL) that we will present in Chapter 7 are evaluated performing *static type-checking.* Static type-checking means to complete type-checking before evaluation, which makes run-time type tests unnecessary. The type synthesis that we perform is comprised of

1. a set of *typing rules,* also called *type system,* and
2. a *type-checking algorithm.*

Before moving to the typing rules and the type-checking algorithm, let us firstly give the SML types that we use for the representation of the expressions and the types of the functional language. Above, we already mentioned that we use VB, VI, VS, VPARMS, VVAR, VAPP, VBRANCH, VFNDEF, VID, VALLIST, and VOP, to represent the definitions and expressions that we have parsed and prepared for compilation. Assume we have defined the SML type system to represent the functional language such that compiled functions are stored by VFUN terms. Furthermore, we use VNULL as a special value to represent "nothing", e.g. the binding of variables which are generated by preparing the parsed expression for compilation.

Thus, the SML datatypes for the representation of statements ("stmt"), values ("value"), and the types of values ("vtype") are given as follows:

```
datatype stmt = FUNDEF of ident * (value * value) list
              | VALDEF of ident * value
              | NULLSTMT
              | QUIT
    and value = VB of bool
              | VI of int
              | VS of string
              | VPARMS of value list
              | VVAR of (string * (value ref * vtype ref)) list ref * int
              | VAPP of value * value
              | VBRANCH of value * value * value
              | VFUN of (value -> value) * vtype
              | VFNDEF of (value * value) list
              | VID of ident
              | VALLIST of value list
              | VOP of string
              | VNULL
    and vtype = bool_t
              | int_t
              | string_t
              | parms_t of vtype list
              | fun_t of vtype * vtype
              | typevar of vtype ref list ref * int
              | noninit_t
  and vtypesc = Forall of vtype list * (vtype ref list ref * int)

 type vartype = vtype ref list ref * int
type varenv_t = string * (value ref * vtype ref) list
```

Type System. For these values ("value") and types ("vtype") we want to explain the typing rules and type inference rules in the following. In the functional language a type is either:

- boolean (bool_t), integer (int_t), or string (string_t),
- or, for tuples, a product type (parms_t),
- or, a type variable (typevar)
- or, a function type $\tau_1 \rightarrow \tau_2$ (represented by fun_t(τ_1,τ_2)), where τ_1 and τ_2 are types.

A type variable (typevar) firstly is bound as an unknown (noninit_t) that we must evaluate to become a more concrete type. E.g., if we evaluate the type of the faculty function, its type is initially unknown (noninit_t), then the type becomes a function type (fun_t(typevar ...,typevar ...)), and finally we obtain the concrete type (fun_t(int_t,int_t)), which is evaluated from the clause that defines the function (respectively from the clauses, in the definition of the faculty function which was last given, and which uses pattern-matching).

Type Environments and Type Schemata. As in [Mau93], a *type schema* is a *type where some variables are distinguished as being generic.* We have implemented type variables and type schemata a little bit different from that given in [Mau93] Chapter 16, here:

1. Type variables consist of a reference to a list of type (vtype) references. We call such a list a *type environment.* In RADD/raddstar different type environments are used. This is for reason to manage the different database schemata and subschemata.
2. In the functional language as well as in the RADD/raddstar specification language, a *type schema* is represented by the `vtypesc` SML datatype, and keeps track of type variables that are equalized to other types or type variables. The most general type of the vartype (`(vtype ref list ref * int)` in the `Forall of ...`), that is, the most general type of the second component of the tuple, is either the typevar of this vartype, in case that the `vtype list` is empty (nil), or else, the head of that `vtype list`.

Typing Rules. A typing rule is written as a fraction where the numerator is called the *premise* and the denominator is called the *conclusion,* and looks like

$$\frac{P_1 \ \ldots \ P_n}{\mathrm{C}}$$

expressing the following: *In order to prove C, it is sufficient to prove* P_1 *...* *and* P_n [Mau93]. If the premise of the typing rule is empty the rule is called an *axiom.* Furthermore, the premises and the conclusions are written as implications

$$\Gamma \vdash \ e : \tau$$

which is read as *under the type schemata* Γ *the expression* e *has the type* τ*.*

- The typing rules for boolean, integer, and string values are axioms.

$$\frac{}{\Gamma \vdash \ \mathrm{VB\ b} : \mathrm{bool\_t}} \ (\mathrm{BOOL}) \qquad \frac{}{\Gamma \vdash \ \mathrm{VI\ i} : \mathrm{int\_t}} \ (\mathrm{INT}) \qquad \frac{}{\Gamma \vdash \ \mathrm{VS\ s} : \mathrm{string\_t}} \ (\mathrm{STRING})$$

- The typing rule for tuples considers the types of the tuple components and generates a product type which is represented as `parms_t` of the list of these types.

$$\frac{\Gamma \vdash \ e_1 : \tau_1 \ \wedge \ \ldots \ \wedge \ \Gamma \vdash \ e_n : \tau_n}{\Gamma \vdash \ \mathrm{VPARMS}[e_1,\ldots,e_n] : \mathrm{parms\_t}[\tau_1,\ldots,\tau_n]} \ (\mathrm{TUPLE})$$

```
(* findtypeenv : vtype -> vtype *)
fun findtypeenv typ =
     let fun findtypeinenv [] =
              raise TypingBug("type not found in env")
           | findtypeinenv (Forall(rl,vt)::l) =
              if typevar vt = typ orelse
               (* if two type vars specifiy the same reference,
                  they are the same *)
               (case vt of (e',p') =>
                    case typ of typevar(e,p) => nth(!e',p') = nth(!e,p))
              then
               case rl of
                  a::_ => a
                | _ => typevar vt
              else findtypeinenv l
     in
      case typ of
         parms_t l => parms_t(map findtypeenv l)
       | fun_t(t1,t2) => fun_t(findtypeenv t1,findtypeenv t2)
       | typevar _ =>
          (findtypeinenv(get_current_env()) handle _ => typ)
       | t' => t'
     end
(* typeofvar : varenv_t ref * int -> vtype *)
fun typeofvar (e,p) =
     (case nth(!e,p) of
         (_,(_,t)) =>
          case !t of
             typevar(e,p) =>
              (case !(nth(!e,p)) of
                  noninit_t => typevar(e,p)
                | t' => t')
           | t' => t')
```

Figure 6.5: The Functions to obtain the most Concrete Type for Type Variables from the Type Schemata.

- Type variables are already assigned a more concrete type, or otherwise, they are represented in the actual type schemata. Like [Mau93], we denote this typing rule *tautology.*

$$\frac{}{\Gamma \vdash \ VVAR(\_,(e,p)) : typeofvar(e,p)} \text{ (TAUT)}$$

  The `typeofvar` function is used to obtain the actual type of the variable, such that– in case it is a typevar at the end of the type inference process –the function `findtypeenv` is applied to that typevar to find the type from the type schemata Γ. Refer to Figure 6.5 for the implementation of these functions.

Generic Instances of Type Variables. A generic instance of a type variable is a substitution of that type variable with a type that is more concrete, but can again contain type variables. E.g., for the first definition of the faculty function

```
fun fac n = if n = 0 then 1 else n * fac(n-1) fi
```

the type inference algorithm assigns the type

```
fun_t('a,'a)
```

```
(* val geninstance : vtypesc -> vtype *)
fun geninstance (Forall(gv,tau)) =
     let fun ginstance (parms_t l) = parms_t(map ginstance l)
           | ginstance (fun_t(t1,t2)) = fun_t(ginstance t1,ginstance t2)
           | ginstance (tv as typevar(e,p)) =
              (case !(nth(!e,p)) of
                  parms_t l => parms_t(map ginstance l)
                | fun_t(t1,t2) => fun_t(ginstance t1,ginstance t2)
                | noninit_t => findtypeenv tv
                | t' => ginstance t')
           | ginstance noninit_t = raise TypingBug"geninstance"
           | ginstance t' = t'
     in
      ginstance(typevar tau)
     end
```

Figure 6.6: The Function to generalize Type Variables in the Type Schemata.

denoting that `fac` has type `'a` and `n = if n = 0 then 1 else n * fac(n-1) fi` has type `'a`. Then, the type of the clause in substituted by `fun_t('b,'b)`, such that the type variable `'a` becomes `fun_t('b,'b)` too. The clause assigns `'b` to the variable n, and from the `n = 0` expression we obtain `'b` as `int_t`.

We finally have to verify that the type of the expressions `1` and `n * fac(n-1)` also reduce to `int_t`. We obtain this way the type of the faculty function, which we did initially set to the type variable `'a`, as

```
fun_t(int_t,int_t).
```

Free and Bound Type Variables. A vartype `vt` of a type schema `Forall(l,vt)` is said *free,* if the list `l` is nil, because then it is not bound to another more concrete type. The same way, if a type variables occurs in `vt`, e.g. if `vt` was `'a` and after `'a` is substituted by another type

```
'a   ==>>   fun_t('b,'b)
```

then `'b` is said *free* in `Forall(l,vt)` if `'b` is not member of `l`. Note, that after the substitution `'a` is not anymore *free*, such that it is said *bound* (namely to `fun_t('b,'b)`). For these aspects we have implemented the functions `varsoftype`, `unknownsoftype`, and `unknownsofenv`

```
val varsoftype : vtype -> vartype list
val unknownsofenv : vtypesc list -> vartype list
val unknownsoftype : vtypesc list * vtype -> vartype list
```

which evaluate the vartypes of a type (varsoftype), all unknowns of the type schemata (unknownsofenv), and the unknowns of a given type (unknownsoftype), respectively. The evaluation of the varsoftype function corresponds to all type variables in a type, the evaluation of the unknownsofenv function corresponds to all free type variables in the type schemata, and the evaluation of the unknownsoftype function corresponds to the free type variables in a type. The `geninstance` function represents the instantiation of a

type schema. This function makes use of the above functions and its implementation is shown in Figure 6.6.

The substitution of type variables, called *generic instantiation* in [Mau93], is considered by the next two rules, where we use the latter two of the discussed functions (unknownsoftype,geninstance).

- The following rule considers the generic instantiation of type variables.

$$\frac{\Gamma \vdash\ e : (\sigma \text{ as } typevar(e,p)) \qquad \sigma' = geninstance(Forall([],(e,p)))}{\Gamma \vdash\ e : \sigma'} \text{ (INST)}$$

- The next rule considers the generalization of a type whose vartypes are bound in the type schemata.

$$\frac{\Gamma \vdash\ e : (\sigma \text{ as } typevar(e,p)) \qquad \alpha \notin unknowsoftype(\Gamma,\sigma)}{\Gamma \vdash\ e : Forall(typevar\ \alpha :: [],(e,p))} \text{ (GEN)}$$

Typing Applications, Conditionals, and Function Patterns. In the list of typing rules and type inference rules above, we still have not considered applications of functions to expressions, if-then-else-fi constructs (conditionals), and the parameters which are introduced after the "fun" and "fn" tokens of the functional language.

- The type of an application of an expression (of a function) e_1 to another expression e_2 is considered by the following rule.

$$\frac{\Gamma \vdash\ e_1 : \text{fun\_t}(\tau_1,\tau_2) \qquad \Gamma \vdash\ e_2 : \tau_1}{\Gamma \vdash\ VAPP(e_1,e_2)) : \tau_2} \text{ (APP)}$$

- The type of a conditional (if-then-else-fi) is given by a boolean expression for the predicate, and by two expressions after the *then* and after the *else* which must be of the same type (τ). Then, the conditional is also of type τ.

$$\frac{\Gamma \vdash\ e_1 : \text{bool\_t} \qquad \Gamma \vdash\ e_2 : \tau \qquad \Gamma \vdash\ e_3 : \tau}{\Gamma \vdash\ VBRANCH(e_1,e_2,e_3) : \tau} \text{ (BRANCH)}$$

- The typing rule for a function which is defined by a single clause only is shown below.

$$\frac{\Gamma \vdash\ e_1 : \tau_1 \qquad \Gamma \vdash\ e_2 : \tau_2}{\Gamma \vdash\ VFNDEF((e_1,e_2) :: []) : \text{fun\_t}(\tau_1,\tau_2)} \text{ (FN1)}$$

- Thus, we can check the type of a function with more than a single clause recursively by the following rule.

$$\frac{\Gamma \vdash\ VFNDEF(c_1 :: []) : \tau_1 \quad \Gamma \vdash\ VFNDEF(l \text{ as } c_2 :: \_) : \tau_2}{\Gamma \vdash\ VFNDEF(c_1 :: l) : unify(\tau_1, \tau_2)} \text{ (FN2)}$$

  The rule evaluates the type which is the most general according the first clause and the rest of the clauses. The unification of the types is done by means of the `unify` function which equalizes type variables and whose implementation is shown in Figure 6.7. Note, that the `unify` function raises a `TypingBug` exception if the types τ_1 and τ_2 can not be unified.

6.2.2.3 From Value Representations to Functions: Compiling the Syntax Tree into Code that can be Immediately Executed.

After type inference and type-checking, the abstract syntax tree of Figure 6.4 can be compiled into code that is executable under a run-time system, which is able to interprete the compiled expressions. In SML, the symbol fn introduces an unnamed function. Thus, the function declarations which are stored as VFNDEF terms are compiled into executable functions (VFUN terms) using the following SML function:

```
fun fnevaluate' (id,_) ([],_) = raise TypingBug"empty function list"
  | fnevaluate' env ((v1,v2)::l,fun_t(t1,t2)) =
     VFUN(fn v =>
           if unify2(v,v1)
            then vevaluate env v2 else fnevaluatep env (v,t1,l),fun_t(t1,t2))
```

As mentioned above, the expression compiler of the RADD/raddstar as well as the *Mini ML Compiler* introduced here are based on λ-calculus. The compile-functions make use of a *fixpoint combinator* (the Z-Fixpoint combinator) for evaluation of recursive function applications. The code of the Mini ML Compiler is shown in Appendix A. The reader who is interested in the design and implementation of functional languages, and their compilation techniques, may also refer to [Mau93], or [Mau95], Chapter 12 - 16, respectively.

```
fun shorten t = shorten'(rtnormalize t)
and shorten' (parms_t l) = parms_t(map shorten l)
  | shorten' (fun_t(t1,t2)) = fun_t(shorten t1,shorten t2)
  | shorten' (tv as typevar(e,p)) =
     (case shorten(!(nth(!e,p))) of
         typevar(e',p') =>
          (shorten (!(nth(!e',p')));
           e :=
            map (
             fn rt => if rt = nth(!e,p) then nth(!e',p') else rt
            ) (!e);
           tv)
       | _ => tv)
  | shorten' t = t
and unify (tau1,tau2) =
     (case (shorten tau1,shorten tau2) of
         (noninit_t,t2) => t2
       | (t1,noninit_t) => t1
       | (bool_t,bool_t) => bool_t
       | (int_t,int_t) => int_t
       | (string_t,string_t) => string_t
       | (parms_t l1,parms_t l2) =>
           if length l1 <> length l2 then
            raise TypingBug("between "^(descrtype(parms_t l1))^" and "^(descrtype(parms_t l2)))
           else
            (let val l1r = ref l1 val ret = ref[] in
              app (
               fn l2e =>
                (ret := (!ret)@[(unify(hd(!l1r),l2e))]; l1r := tl(!l1r))
              ) l2;
              parms_t(!ret)
             end
              handle _ =>
               raise TypingBug("between "^(descrtype(parms_t l1))^" and "^(descrtype(parms_t l2))))
       | (fun_t(t11,t12),fun_t(t21,t22)) =>
           (fun_t(unify(t11,t21),unify(t12,t22))
            handle _ =>
             raise TypingBug("between "^(descrtype(fun_t(t11,t12)))^" and "^(descrtype(fun_t(t21,t22)))))
       | (tv1 as typevar(e1,p1),tv2 as typevar(e2,p2)) =>
           if nth(!e1,p1) <> nth(!e2,p2) then
            (e1 :=
              map (
               fn rt => if rt = nth(!e1,p1) then nth(!e2,p2) else rt
              ) (!e1);
             tv1)
           else tv1
       | (t1,tv2 as typevar(e2,p2)) =>
           if not(occursintype (e2,p2) t1) then
            (set_vartype (e2,p2) t1; tv2)
           else raise TypingBug("between "^(descrtype t1)^" and "^(descrtype tv2))
       | (tv1 as typevar(e1,p1),t2) =>
           if not(occursintype (e1,p1) t2) then
            (set_vartype (e1,p1) t2; tv1)
           else raise TypingBug("between "^(descrtype tv1)^" and "^(descrtype t2))
       | (t1,t2) =>
           raise TypingBug("between "^(descrtype t1)^" and "^(descrtype t2)))
```

Figure 6.7: The Type Unification Function(s).

6.3 The RADD/raddstar Database Type System and the RADD* Data Model

The structural part of the RADD/raddstar database type system and the RADD* data model are based on the functional architecture and the representation of data that we have presented in Section 6.2. The RADD* data model supports most of the features of the data models that we have discussed in Chapter 2 and Chapter 3. Although we are using only a couple of concepts, such that some concepts which are included by the object models presented in Section 3.2 are not considered, we think that the RADD* data model has all the necessary features to perform reasonable and complete database modeling. E.g., RADD* supports all features that we presented in Section 3.3.

Beyond this, the RADD* is used by the RADD/raddstar system to represent the database's maintenance, to evaluate the fitness of the database schema that is under design, and to gather criteria for schema optimization.

In this Section, we introduce firstly the concepts of RADD*'s structural mapping and show then how the integrity constraints are represented. The behavioral part of the RADD* classes stores the method declarations given in the CSL conceptual specification language, that we will look at more closely in Chapter 7. The methods can be added to the structures of the graphical RADD design using the CSL shell, which is the terminal interface of the RADD/raddstar and also part of its graphical user interface (GUI).

6.3.1 RADD* Database Schema and -Structures

The RADD* data model is an extension of the *Entity-Relationship and Behavior Model* (ERBM) which has been presented in [Ste96]. For evaluation reasons, RADD* stores the database structures and operations by terms. The database operations retrieve (select), insert, delete, and update as well as a number of other functions are predefined terms. The predefined database operations can be used with all functions and values which are either predefined or otherwise introduced by the designer using the CSL language. The database structures which are considered, are *attribute, struct, union,* and *path.*

6.3.1.1 Attribute

Attributes are defined by a 3-tuple of a name, a type, and a null-indicator. Functional attributes– class functions & values and member functions & values –are added to the classes of the graphical RADD database design using CSL. If an attribute is representing a CSL member function or value, then the null-indicator is omitted and the "attribute" has a slot carrying that member function or value instead.

Basic ("non-functional") attributes are flat (bool, int, smallint, float, decimal, character, date, time, etc.), structured (records), nested (set, bag, array, vector), or references. Typically, an attribute can have any type that is considered by the RADD* type system (Section 6.3.3). Attributes and composite attributes (records) are constructed using the RADD entity-relationship editor. Tuples (of record-typed attributes) which are specified in the graphical RADD database design, are represented in the RADD* the same way. Consider the "Employee" entity-structure in Figure 4.9 and Figure 3.16.

Figure 6.8 shows the GUI of the RADD/raddstar system and how the "Employee" structure is formally presented to the database designer.

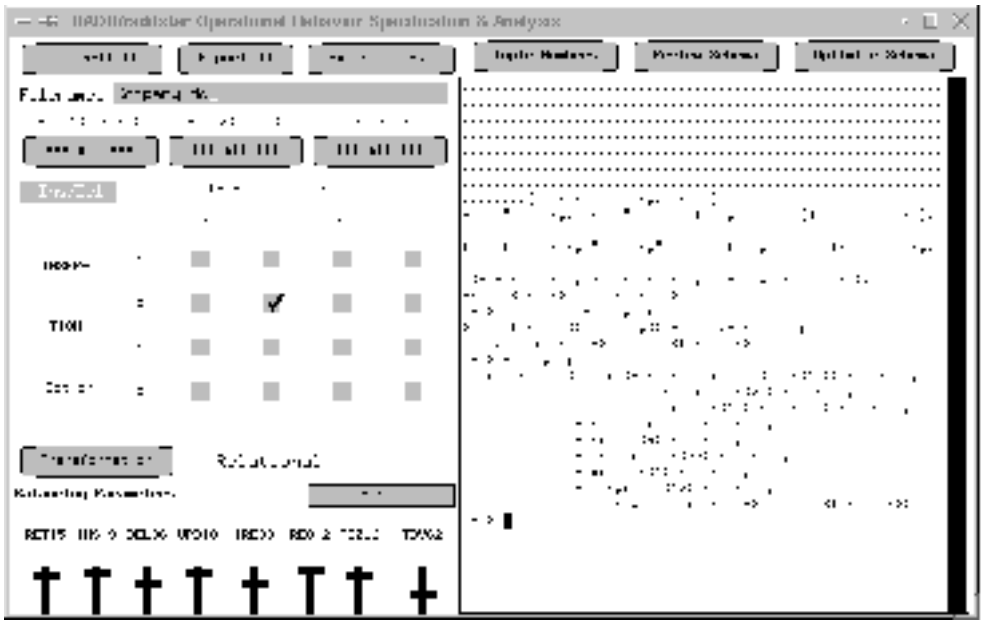


Figure 6.8: RADD/raddstar Representation of the "Employee" Structure.

6.3.1.2 Schema, Struct, Union, and Path

Entity and relationship structures of the graphical RADD database design are represented by RADD* Structs. RADD clusters are represented by RADD* Unions. (Refer to Section 3.3.3.1.) Furthermore, Paths are considered by the RADD*, as they are by the HERM and the RADD data model. Struct, Union, and Path are the building blocks of the RADD* Schema.

Schema. A *schema* is comprised of an id, a set of structures (struct,union), an integrity constraint set, a set of behavior rules, a set of schema functions & values, and a storage parameter describing the default storage organization of the structures. Each of the constraints in the integrity constraint set has references to the structures (or to *paths*

of the structures) of the same schema. The behavior rules are used as default rules for integrity maintaining activities of transactions (*general rules*), e.g.

```
for Ref child->parent: on insert cascade
```

which specifies cascading insert operations: If the target value (foreign-key) is not already contained in the referenced relation ("parent") then an associated record (or object) has to be generated. Behavior rules are otherwise attached to the integrity constraints and structure references of the schema; these latter behavior rules are called *special rules,* and they override the general rules which are attached to the schema.

Struct. A *struct* is an 8-tuple (I,R,S,K,M,T,O,P). I is a tuple consisting of the id of that struct and a (possibly empty) set of ids of components *(in the graphical RADD design, a struct can be nested).* The components are structs or unions. R is a (possibly empty) set of references to other structures. Each element of R consists of

1. a reference to a struct or union,
2. a key-indicator (the struct inherits its key or a part of its key by the reference),
3. a null-indicator,
4. optionally, a role (ISA, PART, MAYBE, or another user-defined ROLE), and
5. a possibly empty set of behavior options and specifications.

S is an ordered set of attribute references (the struct's schema), and K is the key set, such that each element is also an attribute reference. Key-attributes must be in S or in the key-attribute sets of the referenced structures for which the key-indicator is true. M is an ordered set of application module declarations, i.e. CSL class functions & values, which are attached to the "classes" and not to members of the classes. Each element of M consists of a triple which is comprised of

1. the declaration's unique identification that is coded as a string,
2. a precompiled syntax tree of that declaration, where identifiers of the parsed CSL statement are replaced by typed (and possibly bound) variables, and
3. the compiled declaration.

T represents the expected tuple number (number of tuples) of the struct. The tuple number is optional, and it may be 'defined', 'assumed', 'greater than or equal to', 'lower than or equal to', 'greater than or equal to and lower than or equal to', or 'unknown'. Since the tuple number is optional, if it is 'unknown' then it is interpreted as if it were not defined. The O component consists of a tuple of (1.) the physical storage organization used for the primary key, i.e. the default storage which is inherited from the schema or else a special kind: *Heap, ISAM, EHash, . . . ,* and (2.) the physical organization of secondary indices. The storage organizations can be maintained by use of the CSL modify command,

and each storage organization is annotated configuration parameters *(block size, fanout, bucket size, ...)*. P is a set of preceders. Preceders are references to previous structures from which the actual structure is obtained. The conceptual structures of the given schema have no preceders. If a structure is newly generated by the transformation or optimization, the preceders are set in the new struct accordingly.

Union. A *union* is defined by a 4-tuple (I,R,T,P). I is the id of that union. R is a non-empty set of references to other structures (struct or union), each of which is constructed similar a struct reference but omits the null-indicator (the reference is always nullable). T and P are the same as the tuple numbers (T) and preceders (P) of a struct.

Path. *Paths* can be looked at as collections of structures (attribute, entity, relationship, cluster) that are connected some way. In RADD*, an attribute, a structure (struct,union), a projection of a path on another path, a join, a cartesian product, a union (which is not the same as the *union* structure), or a reference of a path to a structure can be a path. Furthermore, if p is a path then p^t denotes the *concrete instantiation* of p at time t. That is, p^t represents the set of p-instances at time t. Concrete instantiations are defined for projections, joins, unions etc. as well (e.g. $p^t[X]$). If u is a member (tuple) of p^t ($u \in p^t$), then $u[X]$ denotes the projection of u on schema X. So, $u \in p^t \Longrightarrow u[X] \in p^t[X]$. A path $p1$ is said *nullable* in path $p2$ if the concrete instantiation of $p1[p2]$ (i.e. the set $p1^t[p2]$) can have a member which contains a $NULL$.

Each structure of the RADD database design workbench is maintained by an RADD* handle (id) which is unique according to the schema. If a structure $s1_x$ of schema $S1$ is equal to a structure $s2_y$ of schema $S2$, then $s1_x$ and $s2_y$ are maintained by the same handle. A schema S must not contain two structures s_x, s_y ($s_x \neq s_y$) such that s_x and s_y have the same handle (id).

In HERM and RADD, a relationship type is said *order-1* if all parent structures are entity types (entity types can be looked at as order-0 relationship types), otherwise the order of that relationship type must be greater than the order of all referenced structures, respectively. The order of cluster types is defined accordingly; a cluster type which has only references to entity types is said order-1, otherwise its order is greater than the highest order of the referenced relationship types.

RADD relationship types are represented by structs with a non-empty set of references in RADD*, RADD *higher-order* relationship types are represented by structs with references to structs which have references in turn. Accordingly, RADD entity types are represented by RADD* structs with no references. It is possible that a RADD* structure of an internally used database schema directly or indirectly (transitively) references itself. This way, the *order* can not necessarily be evaluated for all structures of an internal RADD* schema. This kind of "cyclicity" is not possible for HERM or RADD relationship

or cluster types, nor is it possible for structures of a conceptual RADD* schema, such that the *order* is given for the structures of these schemata.

Key-attributes are represented in the graphical HERM or RADD entity-relationship schema by underlined items (on an empty node-type). For example, a key attribute Employee.Ssn is represented in RADD* by means of a structure 'Employee' with key '{ref ssn}', where 'ssn' is the attribute with name "Ssn".

6.3.2 RADD* Constraints

The definition of KDs, CCs, FDs, IDs, and EDs is equal to that given in [PBGG89, Tha91], besides the fact that left-hand-side and right-hand-side are not related to values only, but can also be containers of complex objects (classes, which are here represented by structs, unions, attributes, references, their combinations etc.).

References (REFs), indicated by arrows between structures in the graphical RADD design, are represented by references of RADD* structs and unions. Cardinality constraints (CCs) which are defined in the graphical RADD design near the arrows are represented by associated RADD* constraints. Functional dependencies (FDs), inclusion dependencies (IDs), exclusion dependencies (EDs), afunctional dependencies (ADs), and path constraints are considered as well. The definition of RADD* constraints is derived from the relational data model and its operators, but, as mentioned above, considers object semantics and pointer semantics as well.

In HERM and RADD, paths are used to specify complex conditions for the database's maintenance. An example of a path constraint is a database consisting of

- entity structures: bus-driver, bus, bus-type, driving-license,
- and relationship structures: drives, has-type, has-license,

such that we can specify the following path constraint:

> The bus-driver who drives a bus must have a driving-license for that bus (for the type [bus-type] of the bus).

Then, this is represented in RADD* by a term like the following:

$$ID(REF(JOIN(REF(JOIN(\text{bus-driver},\text{drives}),\text{bus}),\text{has-type}),\text{bus-type}),\\ REF(JOIN(\text{bus-driver},\text{has-license}),\text{bus-type}))$$

6.3.3 RADD* Type System and Subtyping Rules

In Section 6.2.2, we did describe how the type inference system of the RADD/raddstar, which is based on the λ-calculus, is implemented. In this section, we want to give the reader an overview what types are considered by the RADD* data model, because the type inference system evaluates these types.

6.3.3.1 Properties of the Type System

In RADD*, a type is either

- a denotator type τ, such that τ may be one of the basic types mentioned in 6.3.1.1,
- a record type which is represented by a list of attributes–
  like the schema of a struct (see also Section 6.3.1.1), or
- an arrow type $\tau_1 \rightarrow \tau_2$.

Furthermore, types can be constructed (and are inferred) in the usual way of functional type systems *(product types, sum types).* Variables and expressions of denotator types and types of records which have only attributes of denotator types can be used with the comparison operators = and <>, whereas arrow types can not. Arrow types are used to represent the type of functions and functions can not be compared. The types of references to functions and the types of records which have only attributes of denotator types are also considered as denotator types.[7]

E.g., if a function takes an integer argument and evaluates a boolean, then its type is *int* $\rightarrow$ *bool*. A function may be *curried,* assume

```
substr : string -> int -> int -> string
```

is such a curried function. Then, the expression `substr "myname"` evaluates again a function, which has the type `int -> int -> string`.

The types *int, float, smallint, smallfloat, dec, date, time, money, etc.* describe primitive types that are found in hierarchical, network, and relational databases. Hence, these are represented as axioms in the RADD* type system. The type *character* represents a single character, and the type *binary* represents a single byte in the RADD* type system, respectively. The types *set, bag, array, vector, ref,* and *record* are the type constructors. New types can be created using the type constructors and the types which are already defined. The type inference rules which are used to obtain the types of the functional expressions, derive types in the opposite way.

[7] Denotator types are sometimes called *equality types.* Refer also to [BO96].

6.3.3.2 Type Equivalence Rules and Subtyping Rules

The SQL types *char* and *varchar* need not to be considered separately in the RADD* type system, since they are generated using the type constructors *array* and *vector*. Array and vector have the usual meaning, such that the SQL type "char" is considered as one-dimensional array of the character type, and "varchar" is considered as one-dimensional vector of the character type. A vector's maximal length can be undertermined (possibly infinite). This is then encoded by a -1 length. A one-dimensional vector of infinite length describes what is usually understood as a list. The type (constructor) mappings are described by type equivalence rules, e.g. *[(τ,-1)] vector = τ list.*

Subtyping. W.r.t. the relation $\leq_t$ which is transitive and antisymmetric, the following subtyping rules are considered by the RADD* type system:

- *character* $\leq_t$ *binary*,
- *smallint* $\leq_t$ *int*,
- *int* $\leq_t$ *smallfloat*,
- *smallfloat* $\leq_t$ *float*,
- *dec(len=n,...)* $\leq_t$ *[(character,n)]* array, and
- *'a array* $\leq_t$ *'a vector*.

Unlike other typed functional languages, such as *Standard ML of New-Jersey,* this allows to have functional expressions like `1.2 * 3` which use arguments of different types on the left-hand-side (float) and right-hand-side (int)– although the `*` infix operator of RADD* has the type

```
* : ('a,'a) -> 'a
```

where `'a` is a type parameter, respectively.

6.3.4 RADD* Internal Schema

The constructs we use for the internal schema are an extension of the conceptual schema constructs, and are given as follows.

Definition (Internal Schema). An internal database schema needs not necessarily to be acyclic:

- *Attributes* can additionally have the following types: oid, attr ref, and structure pointer. which denote the usual constructs for a database implementation schema, object identifier and reference to an attribute or structure.
- *Entities* are considered as above, but they are now specified by a 4-tuple (n,A,K,P) where P is a set of references to the preceding (conceptual) structures.

- *Relationships* may directly or transitively reference themselves. For permission of a cyclic data schema, condition (order) is not required for the internal schema. Like entities, relationships are now specified by (n,F,A,K,P).

- *Clusters* are represented by relationships with optional parent structures, i.e. the internal schema does not contain any cluster.

The internal representation is defined by a new structure set and a new constraint set. Actually, we do not distinguish between types for conceptual and internal structures: the precederer component P of each conceptual structure is always the empty set. Also, the constraint types are specified conceptually as well as internally by the same type constructors. Cardinality constraints are specified by a 5-product (P1,P2,c,b,P) where P is a set of references to the preceding constraints, and the other constraint types are represented by a 4-product (P1,P2,b,P) where P– as for the entities and relationships –is empty, for all conceptual constraints, respectively.

6.4 Summary and Outlook

The Chapter gives an overview on the functional concepts of the RADD/raddstar implementation, and on the system's database schema representation.

The RADD* data model which was defined in Section 6.3, is used for the conceptual schema's representation as well as for the internal schema's representation. The behavior options which are specified by the database designer, and the functional declarations which she/he adds to the graphical RADD database design, are either stored by the RADD* schema ("schema" functions & values), or else, directly attached to the structures (classes) of that schema ("class" respectively "member" functions & values).

The following Chapter (Chapter 7) presents the CSL conceptual specification language, which is used to control, invoke, and maintain the activities of the RADD/raddstar system. CSL enables the database designer to add the functional declarations to the graphical RADD database design:

1. In form of transformation rules, general behavior specifications, and schema functions & values, which are attached to the schema,

2. and, in form of special behavior specifications and class and member functions & values, which are attached to the classes and connections.

Chapter 7

Conceptual Specification Language

In Chapter 6 the functional implementation of the RADD/raddstar was described:

- we showed the type inference algorithm– as implemented in the system (Section 6.2), and
- gave an overview on the representation concepts by defining the data model that we are using (Section 6.3).

In this Chapter we will present the specification language of the RADD/raddstar system, called *conceptual specification language* (CSL).

The commands of the CSL can be distinguished in the following groups:

1. session control commands,
2. object specification commands,
3. object description commands, and
4. functional specifications.

The session control commands include the loading of a new database schema, the reviewing and the optimization of the schema, and the invocation of the *tuple number interface*. Beyond these, the user has the ability to set some global variables, such as whether the transformation or optimization of the schema is done interactively or not, or which type of tranformation to use (hierarchical,network,relational,object-relational,object-oriented)– this is specified by the user by means of the `set` command. The latter type of global variable assignment can also be done using CSL reference value assignments (`:=`), but it its more secure to modify these options using the set command, since this way the environment is also set appropriately. Changing these values by the buttons of the graphical user interface– shown in Figure 6.8 –is equivalent to using the set command for their modification.

The object specification commands are the attachment of the behavior specifications and methods to the classes of the graphical RADD database design.

The object description commands are introduced by the `desc` (`describe`) keyword, and are used to describe the structures, integrity constraints, triggers, or transactions, which are defined for the current conceptual schema, the optimized schema, or the schema used for the internal cost evaluations (internal schema). As shown by the `Employee;` command in Figure 6.8, some of these terms– like the structures ('classes') of the conceptual schema –can also alternatively be described by reference to the according CSL value.

The functional specifications comprise the SML[1] like value and function definitions that can be given by the database designer (user). These user-defined values and functions as well as the predefined values and functions are used in the behavior specifications and methods.

The Chapter is organized as follows. Firstly, in Section 7.1 we give an introduction to the property and requirement specification in the RADD/raddstar. Then, in Section 7.2, the functional specification language part of CSL is presented. This part of CSL is an extension of the small functional, SML like language that we presented in Section 6.2. The database programming extensions of CSL are more appropriately considered in Section 7.3, and, Section 7.4 summarizes the Chapter.

7.1 CSL Property and Requirement Specifications

Knowing the information system dynamics supports the process of inductive design decisions. These infomations are important when one wants to model the database system correctly, considering which parts of the schema need to be strongly normalized, and which parts or sets of structures can be collapsed to improve retrieval performance. For these reasons, today there exist a couple of conceptual modeling tools (for relational DBMSs) which allow to model schemata that are either not completely normalized or deliberately denormalized.

However, the decision whether the internal database schema has to be strongly normalized, or if a denormalized internal schema can be used, should not be given at conceptual design time, but informations to support these decisions can already be acquired during conceptual design. Criteria to support these decisions are– for instance –the population sizes of the structures of the designed conceptual schema.

In contrast to what traditional lecture books (e.g. [Vos87, Ull88b, Dat92, RM92]) are teaching about relational database designer and normalization, collapsing structures, e.g. an entity structure and a relationship structure, can improve retrieval performance

[1] SML is used as synonym for Standard ML of New-Jersey, the functional language which we discussed in Section 6.2.1 and with which RADD/raddstar is implemented.

as well as insert, delete, and update performance. E.g. in [Ste95], it has been shown that for the Company schema and its tuple numbers, 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.

The following Section shows how the according design information neccessary for the decision whether to use a normalized of denormalized internal database schema is introduced into and maintained by the RADD/raddstar system.

7.1.1 Maintaining Database Population Information

Whenever, a new *data dictionary* of the RADD entity-relationship editor is loaded into the RADD/raddstar, the system looks for an according *tuple number specification file* at the same place, and– if found –loads the tuple numbers from it. The CSL language is then used to aquire, change, and store the tuple numbers. An interactive dialogue which presents and acquires tuple numbers for the structures of the current conceptual schema is shown in Figure 7.1.

```
CSL> load DD from "Company.dd";
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................[opening Company.tunums]
File "Company.tunums" successfully loaded [3 TupleNumbers].

Data Dictionary "Company" successfully loaded from Company.dd
 (8 Structures, 7 Constraints, 0 Behavior Options).
Press <enter> to continue=>
CSL> enter tuple numbers;

NOTICE: This is a very simple, interactive Menu.
It allows you to enter expected Numbers of Tuples for the modeled Structures. Yo
u may skip the current structure by typing a blank ' ' or zero '0', step back to
 the previous structure by '-1', or cancel the menu by '-2'

Tuple Number for Type 'Department' [4] >>
Tuple Number for Type 'Employee' [760] >>
Tuple Number for Type 'works_for' [=760?] >>
Tuple Number for Type 'manages' [=4?] >>
Tuple Number for Type 'Project' [17] >>
```

Figure 7.1: The Tuple Number Dialogue of the RADD/raddstar.

The dialogue in Figure 7.1 shows that the data dictionary "Company.dd" is loaded into the RADD/raddstar, such that the tuple number specification file "Company.tunums" was loaded as well. The user entered then the CSL command `enter tuple numbers`– to maintain the tuple numbers of the Company schema. Here, the line

```
Tuple Number for Type 'works_for' [=760?] >>
```

indicates, that the tuple number of "works_for" (`[=760?]`) was not defined in the "Company.tunums" file, but can be inferred from the tuple numbers defined there and the integrity constraints which are given for the schema.

7.1.2 Deriving and Advising Schema Transformations

The graphical RADD conceptual design schema does not already contain information how to realize the database internally. That is, this schema posseses the following properties:

1. By the HERM order requirement (Section 3.3.3.1) the graphical schema does not contain mutual references or cycles, neither explicite nor implicite.
2. The graphical schema is usually completely normalized, since the database designer need not to use repeating groups etc.– for reason that list, set, and record-typed attributes are supported by the HERM and RADD data model.

However– as mentioned in Section 6.3.4–, a schema that is generated by the RADD/raddstar transformation, to evaluate the behavior and performance of the conceptual schema, may be cyclic and can have repeating groups as well (in case of a relational transformation). Furthermore, the transformations which are performed do not only refer to the transformation rules, but also to the transaction and behavior properties 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 (collapsed), since we assume that the user wants to see the actions 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 Ref works_for->Employee:
>     on insert cascade;
Adding 1 new rules to schema Company.
```

From this behavior rule specification, RADD/raddstar assumes that the designer does not want the structures *Employee* and *works_for* to be collapsed on the internal schema: *The specified* `on insert cascade` *would never take place since it were given implicitely all the time, by the collapsed internal structure (Employee,works_for).*

On the other hand, it is not convenient to control transformation processes only by behavior specifications, Therefore, we give the database designer also the opportunity to explicitely forbid transformations:

```
CSL> add transformation rule:
>     do not group(Employee,works_for);
Adding new global transformation rule as "g1".
```

In the example, the database designer explicitely specifies a "global" transformation rule (which was named "g1", by the system) that disallows to collapse *Employee* and *works_for* to one internal structure, by the transformer of the RADD/raddstar. The designer may also specify rules, which advise the transformer to do transformations that are not given implicitely.

7.2 CSL Functional Specifications

In Section 7.1 we gave some examples of adding properties and requirements to the datbase schema using CSL. Besides the *restrict, cascade, set null, and set dafault* behavior options which we already presented in Chapter 5, RADD/raddstar allows also to use *behavior specifications* with the help of user-defined database functions and IF-THEN-ELSE-FI statements, such that transactions can be specified in a programmatical way. The values and functions that are defined in the SML like language (which is an extension of that introduced in Section 6.2) can be used as functional values per se, and can also be attached to the classes of the graphical RADD database design.

Generally, CSL provides the following term and function application constructs for enriching the conceptual schema with database application semantics:

- The terms *select, insert, delete,* and *update* are used as denoters for database operations– RADD/raddstar uses the so constructed operation terms of the behavior and transformation rules– e.g. *(insert,Employee)* or *(insert,Employee{Name="Victor H.", Bdate="11-01-59", Salary=54000, ...})*, to evaluate transactions and necessary sub-transactions.
- The functions *entity, relationship, cluster, component* and *tcomponent* are used as testing operations for the item of the data schema; e.g., if *Employee* is an entity type then *entity Employee* returns true, or if *manages* directly or transitively references *Employee* then *tcomponent manages Employee* returns true.
- The functions *compatible, highcomplexity* and *attrsize* are used as property evaluators.
- The functions *group, separate, nest, unnest* and *clusterize* are the schema transformation operations.
- And, as introduced in Section 6.2, the operators "+", "−", "∗", "/", "˜" (unary minus), and "^" (string concatenation) are the primitive operators, "=", "<", "<=", ">=", ">", and "<>" are the comparison operators, and ":=" is the assignment operator.

The syntax of CSL functional expressions is similar the syntax of the small functional language given in Section 6.2.2, and has additionally the following characteristics:

1. The infix operators "+", "-", "*", and "/" have the signature `'a * 'a -> 'a` , such that the type allows them to be used for integers as well as for floats.
2. The type system is more flexible and richer than that of the functional language of Section 6.2.2. So, traditional database types like the SQL-type DATE, and collection types (set, bag, list, array, vararray) are included.
3. *Reflective Type System.* Like RDBMSs which use some special tables as a *catalog* to describe the structure of the database, in the RADD/raddstar the structures of the conceptual schema are abstract data types (classes) which are represented as CSL values. It follows that the *Employee* entity type of the Company schema which is maintained as a class, can be used in any CSL expression as if it were a value. So, terms like *Employee*– or the attributes of the classes, like *Birthdate* –can be used as constant values in the definitions of new values and functions, and the specifications of transactions, transformations etc.
4. The symbol "this" is used to refer to the current object of a class. E.g., if a member funcation is attached to the *Employee* class, then `this.Birthdate` can be used to refer to the *Birthdate* of the current *Employee*.

7.2.1 Defining and Using Application Functions

Values, i.e. constants, can be defined in CSL by expressions of the form:

'val' < *ident* > '=' < *cslval* >

and functions by:

'fun' < *value* > < *par* > '=' < *cslval* > { '|' < *par* > '=' < *cslval* > }∗

where < *cslval* > is a constant, a CSL-defined value, or a CSL expression:

cslval ::= < *const* > |
< *ident* > |
'fn' < *par* > '=>' < *cslval* > { '|' < *par* > '=>' < *cslval* > }∗ |
'let' <decl>∗ 'in' <cslval> { ';' <cslval> }∗ 'end' |
'(' <cslval> { ',' <cslval> }∗ ')' |
< *cslval* > < *op* > < *cslval* > |
< *cslval* > < *cslval* >

Like already mentioned, the syntax of CSL is an extented form of the syntax of the functional language given in the *Parsing* paragraph of Section 6.2.2. Note, that like in SML the symbol 'fn' introcuces the definition of a function which has no own identification and is therefore handled as a value only. In CSL, the primer application of the 'fn' symbol is to specify patterns when defining *curried* functions.

An example of a (predefined) curried function is given by the *group* function:

```
CSL> group;
it : (struc,struc) -> (int,int) -> struc = <function>
CSL> val Emp = Employee;
Emp : struc = {Salary: float not null,
               Sex: char(1) not null,
               Address: varchar(30),
               Name: {Firstnames: list(varchar(15)) not null,
                      Lastname: varchar(20) not null,
                      Title: varchar(10)} not null,
               Birthdate: date not null,
               Ssn: decimal(9) not null}
CSL> group(Emp,works_for);
it : (int,int) -> struc = <function>
CSL> it(0,1);

it : struc = {Salary: float not null,
              Sex: char(1) not null,
              Address: varchar(30),
              Name: {Firstnames: list(varchar(15)) not null,
                     Lastname: varchar(20) not null,
                     Title: varchar(10)} not null,
              Birthdate: date not null,
              Ssn: decimal(9) not null,
              Department: struc ref}
```

Here, the expression `group(Emp,works_for)` evaluates a new function that takes a tuple of type *int∗int* as argument and evaluates a *struc* (class) in turn. The attributes of the class that are generated by the *group* (Employee,works_for) (0,1) are shown after the `it : struc =` line. The type of all structures is "struc". By the `val Emp = Employee` command the type of the *Employee* class is described, and RADD/raddstar presents then the type of the *Employee* class to the database designer as a "value" of the following form:

```
{Salary: float not null,
 Sex: char(1) not null,
 Address: varchar(30),
 Name: {Firstnames: list(varchar(15)) not null,
        Lastname: varchar(20) not null,
        Title: varchar(10)} not null,
 Birthdate: date not null,
 Ssn: decimal(9) not null}
```

The concrete structure of *Employee* which is maintained by RADD/raddstar is considered if expressions (functions,values) which use the *Employee* class are evaluated.

Besides "struc", RADD/raddstar types are "unit", "bool", "int", "real", "number", "date", "string", "data_schema", the generic types "tuple", "list" and "set", and "function". RADD/raddstar maintains the RADD SQL-2 types *char*, *float* and *decimal* by "string", "real" and "number", respectively. The RADD/raddstar internal description of the database attribute types, e.g. $list(varchar(15))$, is converted to the according representation, e.g. $varray\ of\ varchar(15)$, when the database schema is exported to an external language or SQL-dialect, such as Oracle8.

7.2.2 Describing Database Operations

The *group* function of the above examples is a database schema transformation operation. As shown, such CSL functions can be used in other functional expressions as well. RADD/raddstar's type evaluation and inference for CSL functional expression takes care that only complete and type-correct applications are used. Complete and type-correct applications are considered whenever the database designer adds application code to the functions of the graphical RADD schema as well.

Consider the following fragment where the user adds an *age* function to the *Employee* class:

```
CSL> today;
it : unit -> date = <function>
CSL> today();
it : date = 1999/12/15
CSL> add class Employee:
>     fun age() = today() - 20;
Typing error in function age:
 (date,date) -> date applied to ((unit -> date) -> unit,int)
CSL> add class Employee:
>     fun age() = today() - this.Birthdate;
this.age : unit -> date = <function>
```

By means of adding operations to the classes of the graphical RADD design, the user has also the possibility to define *insert*, *delete*, *update*, and *retrieve* (select) operations, and so, to overwrite the default operations which we assume to be given by the DBMS. This corresponds to the CREATE RULE pl_ins AS ... statement in Figure 3.14 and the CREATE VIEW ... INSTEAD OF ... view inserts which we did mention in the Oracle8 paragraph of Section 3.3.2.3.

Application functions and behavior of database operations. If an *insert*, *delete*, *update*, or *retrieve* operation is "redefined" by means of a class method with the same name, then the default behavior rule or behavior specification of the database designer (assume *on-insert-cascade* for the CC($works\_for$,*Employee*)) is omitted by the RADD/raddstar transaction evaluator, whenever the redefined operation includes the according triggered operation (e.g., if *Employee.insert* contains $insert(works\_for)$). Furthermore, in such a case the $insert(works\_for)$ operation is assumed never to trigger $insert(Employee)$ in turn.

7.3 CSL Control Structures and Database Application Programming Extensions

In this Section, we give an overview on the conceptual specification language (CSL) which is RADD/raddstar's user interface.

7.3.1 Syntax of the CSL Commands

CSL has the following control statements:

ctlstm ::= load { DD | CSL } from <*string*> ||
export { DD | CSL | SQL | form } for <*subschema*> to <*string*> ||
load tuple numbers from <*string*> || export tuple numbers to <*string* > ||
enter tuple numbers || define <*constraint*> || <*behavior-rule*> ||
drop rule (<*behavior-rule*>) || drop rule <*string*> ||
add [<*schema-type*>] { transformation | optimization } rule : <*when-rule*> ||
transform <*subschema*> || review <*subschema*> || optimize <*schema*> ||
replace <*subschema*> with <*cslval*> || describe <*string*> ||
set <*ident*> = <*cslval*> || add class <*ident*> : [member] <*csldef*> ||
modify schema to <*storage*> || modify class <*ident*> ["{"<*attrs*>"}"] to <*storage*> ||
<*csldef*> || <*cslval*> ||
help || quit

behavior-rule ::=
for <*constraint*> : <*behavior*> ||
operation (<*databas-operation*> , <*structure*>) is [not]
{ frequently_required | rarely_required | of_high_priority | of_low_priority }

when-rule ::=
when <*condition*> do <*action*>

7.3.2 Semantics of the CSL Commands

The 'help' and 'quit' commands are used for giving a help dialogue and ending the CSL session, respectively. The token <*csldef*> denotes CSL function and value definitions, CSL values (<*cslval*>) are arithmetic expressions, function calls, or simply, bound identifiers; the syntax of which is close to Standard ML ([Mil87]). An example of a CSL function definition is given by the following clause in the style of λ-abstraction ([Mau93], Ch. 12 - 16):

```
fun fac n = if n = 0 then 1 else n * fac(n-1) fi
```

Alternatively, this (schema) function can be defined using pattern matching:

```
fun fac 0 = 1 | n = n * fac(n - 1)
```

In both examples, the `fac` symbol appearing on the right-hand-side of the clauses (bodies) is bound to the identifier after the `fun` symbol (parameter list). Recursion has been implemented using a fix-point combinator.

In a CSL value definition, a symbol appearing in the body is not bound to the symbol that is defined on the left-hand-side, e.g.:

```
val fac = fac
```

binds `fac` to the value (or function) `fac` that must have been defined previously. If a symbol in the body is not bound, an exception is raised (SUnbound of string).

The upper 'load' commands are used to load a *data dictionary* of the RADD schema editor (".dd") or CSL specification file (".csl") into the RADD/raddstar.[2] Here and in the other commands, the filename (<*file*>) must be a string which is enclosed by quotes, such as "Company.dd" or "Company.csl". The associated 'export' commands are used for exporting data dictionaries and CSL files from RADD/raddstar. Here, a complete schema or a subschema of the conceptual, optimized, or internal schema can be exported. The schema is described by the optional word 'conceptual', 'internal', or 'optimized', followed by the string 'schema' or a subschema specification. A subschema specification begins with the string 'SubSchema' and a left curly bracket ('{') followed by a comma-separated list of identifiers,[3] and is closed by a right curly bracket ('}'). By means of the 'export' 'form' command, it is possible to export Web (HTML) forms for subschemata, by which database relations can be retrieved, inserted, deleted, and updated using an interface to a DBMS, which maintains a sample database for the design.

The 'load', 'export', and 'enter' 'tuple' 'numbers' commands are used to load, export, and define (or modify) the tuple numbers of the current conceptual schema. For the current conceptual schema the database designer may want to specify additional integrity constraints, which can be done with the help of the 'define' command. With the help of the 'for' command, he can attach behavior rules to the constraints which are contained in the graphical conceptual schema, or which he defined by means of the 'define' command. In this context, the constraint (<*constraint*>) may be 'all' 'constraints', 'all' 'FDs', 'all' 'IDs', ..., or a special constraint of the current schema (KD,REF,CC,FD,ID,ED,AD).

By the 'for' command, he can attach behavior rules to the constraints which are already contained in the graphical conceptual schema, or which he defined by the 'define' command. In this content the constraint (< *constraint* >) may be 'all' 'constraints', 'all' 'FDs', 'all' 'IDs', ..., or a special constraint of the schema (KD,REF,CC,FD,ID,ED,AD). The behavior rules (< *behavior* >) have the form shown in Figure 5.1. That is, for instance,

on update parent do if nullable(child) then set null else set default fi .

[2]The data dictionaries of the RADD schema editor are normally loaded at invocation time of RADD/raddstar; then, '-LOADDD' and the filename are given as command line arguments.

[3]The RADD/raddstar compiler checks that the identifiers which are used in the subschema specification are structures of the schema.

When the user specifies a new behavior rule for an integrity constraint, then the (possibly empty) list of previously specified rules for that constraint is appended by the new behavior specification. When the database designer uses the 'drop' to delete a behavior rule, then this behavior specification is deleted from the list. The last behavior rule of that list is namly always considered when evaluating transactions for the database schema, such that, for instance, an *on-insert-cascade* covers a previously specified *on-insert-restrict* for the same constraint. But, after dropping the *on-insert-cascade* for that constraint, the *on-insert-restrict* is valid again. In the 'drop' 'rule' command, $< behavior - rule >$ denotes the combination of the integrity constraint and the behavior rule as specified by the 'for' command, e.g.

drop rule (for Ref manages-¿Department: on insert cascade) .

When the user specifies a new behavior rule for an integrity constraint, then the (possibly empty) list of previously specified rules for that constraint is appended with the new behavior rule. When the database designer uses the 'drop' 'rule' command to delete a behavior rule, this behavior rule is deleted from the list. The last behavior rule of that list is always considered when evaluating transactions for the database schema. This way, an *on-insert-cascade* covers a previously specified *on-insert-restrict* for the same constraint. But, after dropping the *on-insert-cascade* for that constraint, the *on-insert-restrict* is valid again. In the first 'drop' 'rule' command, the <*behavior-rule*> denotes the CSL rule that has been defined previously. This can be a maintenance rule as specified by the 'for' command, or else an operation property, which is defined by the 'operation' command. The second 'drop' 'rule' command is used to drop a transformation or optimization rule (*when-rule*), which was assigned a unique name by the system (<*string*>).

The 'describe' is used to describe the structure of objects of the schema, and the 'set' command is used for assigning global variables of the RADD/raddstar, such as the kind of transformation or optimization (interactively or not). By means of the 'set' command also the type of the transformation rules to use and the cost model can be set. For each transformation rule set, 2 - 5 cost models are applicable.

By means of the 'add' 'class' command, the designer can add class functions & values or member functions & values to the classes (remember the description of attributes in Section 6.3.1.1). Member functions & values are automatically recognized, either when the keyword 'member' prefixes the function or value definition or else when the keyword "this" is used in the body of the definition, The keyword "this" is also the only way to call the member functions & values from other function or value definitions for that class. Assume, a member function "age" is added to the Employee class

```
CSL> add class Employee:
>     fun age() = this.Birthdate - today();
this.age : unit -> date = <function>
```

Thus, this.age can be used only by other member functions & values of Employee, or by database operations (retrieve, insert, delete, update) which are working with the Employee set. A member function or value can be accessed (and is protected) the same way an attribute is.

A member value which is added by means of the 'add' 'class' command is like an attribute, despite the following exceptions:

1. The member value has no null-indicator, such that it is always *not null.*
2. Whenever the body of the value definition clause specifies a constant (and not an attribute, like this.Birthdate), the member value is a fixed value that can not be changed anymore, and then it also is the same for all objects of the class.

An exception of this non-changability of member values which are constants may be seen in the use of references; although these are also static values in ML (and so in CSL), their content can be changed every time using the ":=" operator. The content of a reference value, which has possibly been changed, can be retrieved using the "!" operator. Member attributes which are references can be used as communication channel by the objects of a class (or between objects of different classes), since as soon as one object changes the reference value's content all other objects see that new content ("state").

The 'modify' commands are used to change the storage organization which is used as the default for the classes of the schema, which are not assigned a special primary storage organization, to modify the primary storage organization of a class *(i.e., the primary storage organization of the class' member set)*, or to define and maintain additional secondary indices. These are then represented appropriately on the internal schema, and, for the internal schema, SQL schema definition code can be directly generated using the 'export' 'SQL' command.[4] The following gives an example.

```
CSL> export SQL for optimized schema to "../../TeX/tmp/Company.sql";
done.
```

Here, the file "Company.sql" is written in the directory "../../TeX/tmp".
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 specifications 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* foreign-keys as well as for the use of the stored procedures.

[4] Actually, the conceptual or optimized schema as is can be exported to SQL, and also the sets of hierarchical, network, and object-oriented transformation rules can be used to generate an internal schema which is exported to SQL.

This way, the system generates on the basis of a graphical conceptual database design and the logical definition of procedures and database functions (CSL interface) a complete database schema for implementation under a special DBMS, that contains important realization hints. An excerpt from the generated SQL code for the (optimized) Company schema is shown in Figure 7.2.

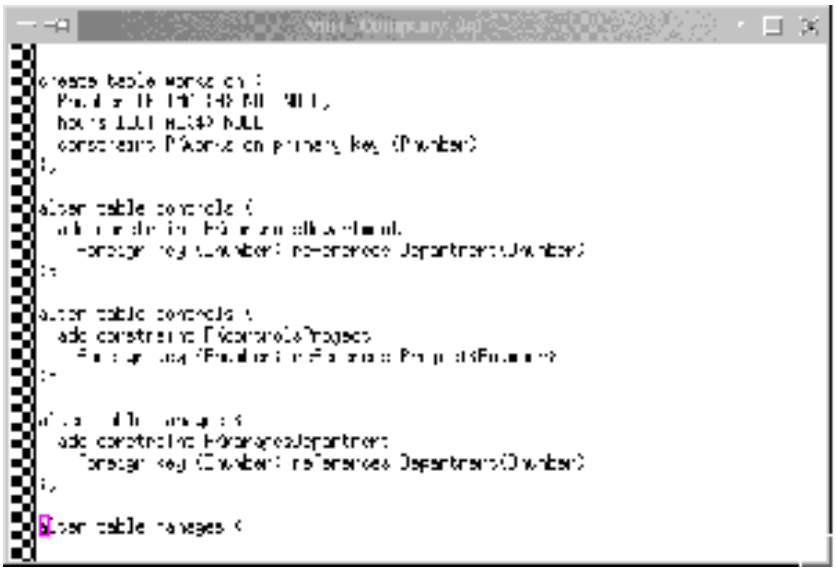


Figure 7.2: SQL Schema Definition Code generated by the RADD/raddstar.

7.3.3 Database Schemata and their Subschemata

At the beginning of Section 7.3, we have mentioned how subschemata are desribed in CSL. In this Section we will explain how they can be used in the RADD/raddstar. Let us consider how database schemata are transformed, reviewed, and optimized firstly.

7.3.3.1 Transforming the Conceptual Schema (The Transformer)

The conceptual schema (or any particular subschema, of the conceptual, the current internal, or the optimized schema) is transformed by means of the 'transform' command. According the chosen schema or subschema, the *transformer* of the RADD/raddstar selects the set of transformation rules for that schema type, and performs either an interactive or a non-interactive transformation of the schema.

The transformation operations which are used in the predefined transformation rule sets, are the group, separate, nest, unnest, and clusterize operation which were described in Section 5.2.2. Figure 7.3 shows, how the predefined rules for relational transformations are defined.

```
(* Transformation rule "r1" *)
add relational transformation rule:
 when CC(s1,s2) is (m,n)
  and component s1 s2
  and m >= 1
  and (n = 1 or (attrsize s1 (m,n)) < !rMaxRepGrpSize)
 do
  group (s2,s1) (m,n)
;
```

```
(* Transformation rule "r2" *)
add relational transformation rule:
 when CC(s1,s2) is (0,1)
  and component s1 s2
  and emptySchema s1
 do
  group (s2,s1) (0,1)
;
```

Figure 7.3: CSL Startup Code to Initialize the Set of Relational Transformation Rules.

The two transformation rules shown in Figure 7.3 are examples for the 'add' (transformation rule) command of CSL. After it has been started, RADD/raddstar looks firstly for an initialization file ".raddstar". If it finds this file, either in current directory or in the user's home directory, then it loads its default transformation rules and behavior specifications from that file. Subsequently– if it is invoked with "-LOADDD" $< file >$ on the command line– RADD/raddstar loads the data dictionary specified by the $< file >$, e.g. "Company.dd". As mentioned in Section 7.1.1, with the "Company.dd" also a tuple number file "Company.tunums" is loaded whenever it exist in the same directory as the "Company.dd". After the tuple number file, a CSL specification that exists in the same directory is loaded; in the example of "Company.dd" this were "Company.csl". The transformation rules and behavior specifications in the loaded database schema's specification file ("Company.csl") override the rules and settings which are given by the ".raddstar" file. And, of course, the ".raddstar" file and the "Company.csl" file may contain again 'load' 'CSL' commands, e.g. the ".raddstar" file can contain commands for loading general transformation rules from other CSL files. The specification files for the optimization rules have the same structure as those with the transformation rules.

CSL forbids to use 'load' 'DD' commands in CSL files: this type of command is allowed only for interactive usage. And, as mentioned above, the type of the performed transformation and the decision whether the transformation is interactive or not, depend on some global variables (typeOfTransformation,kindOfTransformation) that can be assigned by the user (with help of the assignment operator, ":="), but should better be set by means of the 'set' command.

7.3.3.2 Reviewing the Conceptual Schema (The Reviewer)

After the database designer (user) has possibly defined additional constraints and maintenance rules for the graphical entity-relationship schema, the schema fitness evaluation ("reviewing") can be invoked. This is done by means of the 'review' command, which includes a non-interactive transformation in case that the conceptual or optimized (conceptual) schema is reviewed– to evaluate transactions. If the internal schema is reviewed the schema is not transformed by the review process. The *reviewer* displays an X-window which shows the transactions of the (conceptual) schema, their costs, and, if the user

clicks on the cost term with the middle mouse button the according transaction's sub-transactions and their cost terms.

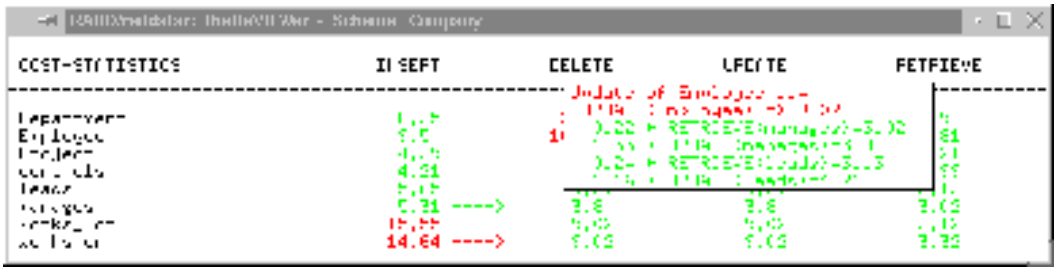


Figure 7.4: Matrix presenting the Conceptual Schema Transactions and their Costs.

Figure 7.4 shows how the reviewer presents the transaction cost matrix to the user. Here, the costs which are assumed to be bottlenecks of the schema are red, whereas the others are green. By use of the right mouse button the user has the possibility to unmark the assumed bottleneck (which changes the color of the cost term from red to green), or otherwise, to mark costs as bottlenecks that are not "high", and so, that are not assumed to be bottlenecks by the RADD/raddstar (then, the color of the cost term changes from green to red). According the diffferent database operation types (insert, delete, update, and retrieve), and according the "bottlenecks" of the previous schema of the optimized conceptual schema, the thresholds which make the operational costs to bottlenecks that are presented to the user (red) or not (green), are different.

7.3.3.3 Optimizing the Conceptual Schema (The Optimizer)

The optimize command can only be invoked for a complete schema, such that as the $<$ *schema* $>$ token of this syntax rule only a schema and not a subschema must be used. That is, the word 'schema', which may be prepended by the word 'conceptual', 'internal', or 'optimized'– like in the subschema specification. If 'conceptual', 'internal', and 'optimized' are omitted, then the conceptual schema is optimized.

The *optimizer* considers the bottlenecks which are finally marked in the cost matrix that is displayed by the reviewer. E.g., the *highcomplexity* function of the optimization rule used in Section 4.2 (Figure 4.13) returns true if and only if the operation is marked as bottleneck in the reviewer's cost matrix. This way, if the user unmarks a bottleneck, then this is also not considered as a bottleneck by the schema optimization process. Besides defining own, special schema optimization rules, this is the only place where the user has influence to the schema optimization process.

7.3.3.4 Subschemata

As described above a subschema specification is beginning with the string 'SubSchema', and has then a left curly bracket ('{'), a list of identifiers which are checked– by the CSL compiler –to be structures of the current conceptual schema, and finally, a closing right curly bracket ('}').

```
CSL> group(Employee,works_for) (1,1);

it : struc = {Name: {Firstnames: list(char(15)) not null,
                     Lastname: char(20) not null,
                     Title: char(10)} not null,
              Birthdate: date not null,
              Ssn: decimal(9) not null,
              Address: char(33),
              Sex: char(1) not null,
              Salary: decimal(12) not null,
              Department: struc ref}
CSL> replace SubSchema{Employee,works_for} with it;
Transformer: 1 transformations have been applied to the conceptual schema.
```

Figure 7.5: Substituting the Subschema {Employee,works_for} by the grouped Structure.

Modifying Subschemata. The modification of a subschema is done by means of the 'replace' command. By the replace command a subschema (or schema) is substituted by another (sub-) schema, such that the references are set appropriately. Figure 7.5 gives an example how the {Employee,works_for} subschema is replaced by the grouped (Employee,works_for) structure.

Generating Web (HTML) Forms for Subschemata. With the help of a subschema specification it is possible to modify particular parts of the conceptual, internal, or optimized schema, and also, to generate Web (HTML) forms which are providing a user-interface to the database.[5] E.g., an HTML form for the {Employee} subschema (the Employee structure) is generated by the following command:

```
CSL> export form for SubSchema{Employee} to "../tmp/e1.html";
File "../tmp/e1.html" exists! Do you want to overwrite [N/y] ? y
OK - overwriting file ../tmp/e1.html
done.
```

The form which is generated by that export command is shown in Figure 7.6.

[5]The select, insert, delete, and update operations which are available with the "Retrieve", "Save", "Delete", and "Update" button of this mask will be posted to a PostgreSQL database. The interface that provides this functionality is currently in work.

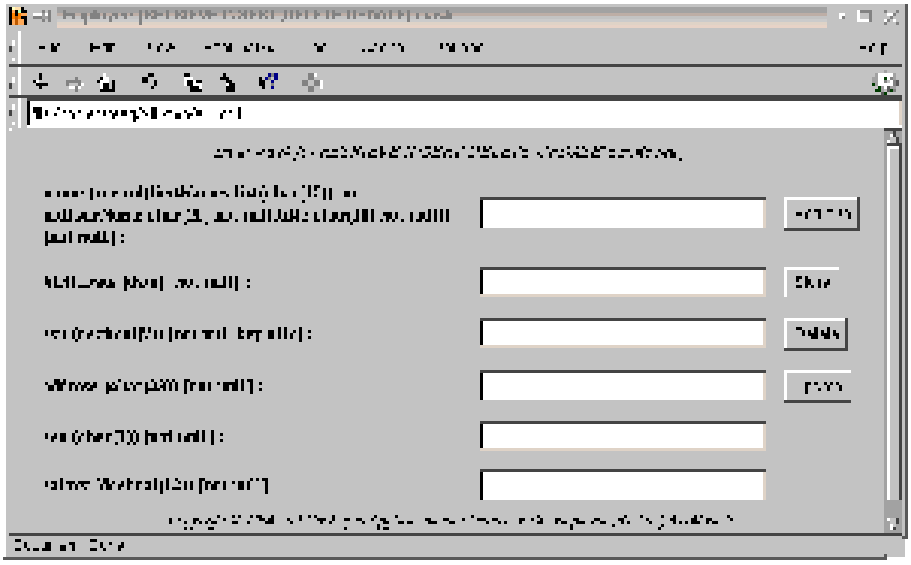


Figure 7.6: Exported HTML Form for the {Employee} Subschema.

7.4 Summary and Outlook

The Chapter presented the syntax of the design specification language that is provided by the RADD/raddstar, called conceptual specification language (CSL). The CSL commands may be distinguished into the following classes: session control commands, object specification commands, object description commands, and functional specifications. For reason that the different classes of the CSL commands are interrelated, we did use a separate Section for each class, but presented them in their context.

1. For the *session control commands* and the *object specification commands* we gave some examples in Section 7.1 and considered them again in Section 7.3.
2. And, the *object description commands* and the *functional specifications* were considered in Section 7.2.

In the following Chapter we will present a complete specification and optimization scenario of the latter Company schema– which was shown in Figure 3.16. We will also give some additional explanations for using the graphical interface of the RADD/raddstar, which was shown in Figure 6.8.

Part III

Conceptual Database Design Optimizer

... knowing the relative importance of the various transactions and the expected rates of their invocation plays a crucial part of physical database design. It is true that only some of the transactions are known at design time. After the database system is implemented, new transactions are continously identified and implemented. However, the most important transactions are often known in advance of system implementation and should be specified at an early stage.

R. Elmasri and S.B. Navathe,
in [EN89].

Chapter 8

Conceptual Database Design Optimizer

In Section 3.3.3.1 and 3.3.3.2 we gave an overview on the HERM and RADD data model, which serve as basis for the RADD/raddstar internal data model. The RADD/raddstar internal data model (RADD*) was defined in Section 6.3. In Section 6.2 we also introduced the basics of the functional compilation kernel, which serves for the type-checking and evaluation of the CSL functional specifications that we considered in detail in Chapter 7.

Now, this Chapter describes the system architecture of the RADD/raddstar and presents an application scenario of the RADD/raddstar. The application scenario shows how the *conceptual database design optimizer* interacts with the designer and supports him in specifying additional semantics and requirements, and in improving the schema.

The Chapter is organized as follows. Section 8.1 gives an overview on the components of the RADD workbench, and especially on the system architecture of the RADD/raddstar. Section 8.2 shows how additional requirements are specified for the graphical RADD database design. Section 8.3 illustrates how the RADD/raddstar reviews the schema, marks probable bottlenecks, and how the bottlenecks can be adjusted by the database designer. It shows further how raddstar optimizes the database schema, considering the additionally introduced requirements and marked bottlenecks. Section 8.3 also demonstrates how the finally optimized database schema is made available for reload into the graphical RADD schema editor, and how the tuple numbers and CSL specifications are stored in files for later use.

8.1 System Architecture of RADD and raddstar

Figure 8.1 shows the system architecture of the RADD *(Rapid Application and Database Development)* workbench and its subsystem raddstar (RADD/raddstar). The magenta box contains the components of the raddstar, and the blue box contains the other RADD components. The green box shows the files which are used for interaction between RADD and raddstar.

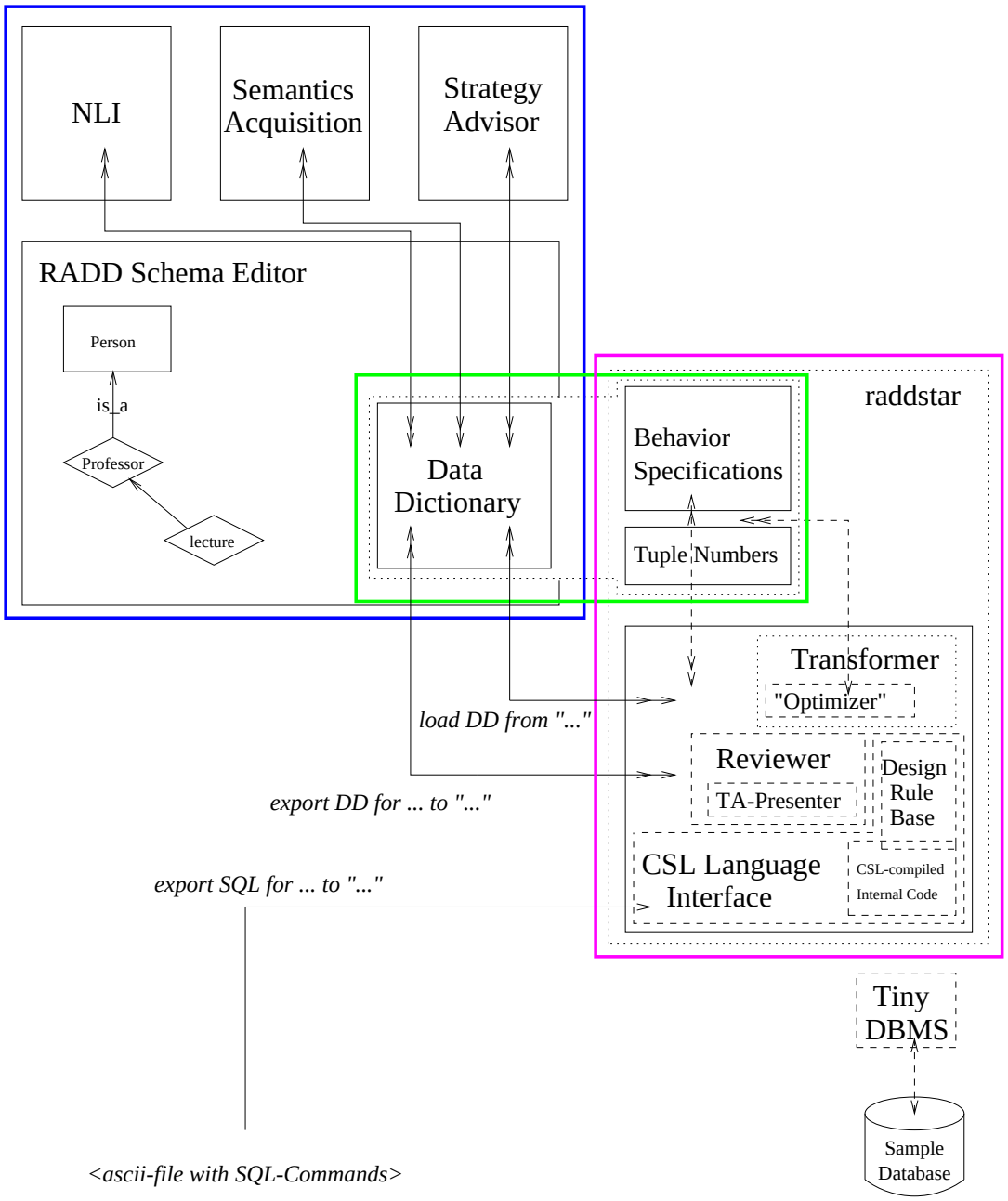


Figure 8.1: The RADD Workbench and its Subsystem raddstar (RADD/raddstar).

The *data dictionaries* of the *RADD schema editor* (the editor used for the graphical entity-relationship design) serve as the communication interfaces between the following components of the workbench:

- the RADD Schema Editor,
- the Natural Language Interface (NLI),
- the Semantics Acquisition,
- the Strategy Advisor, and
- the RADD/raddstar.

The *RADD schema editor,* the *natural language interface,* the *semantics acquisition,* and the *strategy advisor* are described in other publications, e.g. [BDT94, Alb94, AAB+95], such that we do not describe them in further detail here.

In Figure 8.1 a RADD subsystem named *Tiny DBMS* (TDBMS, and a *Sample Database*) is shown as well. The TDBMS once was a component of the RADD/raddstar, which could be used to prototype the user's database design, declarations, and specifications in a "real" database environment, using an SQL-like database language extension of CSL. E.g., in [AAS97a] and [AAS97b] examples were given how the classes of the graphical design and database views can be combined– using the TDBMS interface. However, we deleted the TDBMS again, because there were conflicts with the type inference system of the CSL compiler, and this DBMS functionality is now replaced by the PostgreSQL database interface mentioned in Section 7.3.3.4. For reason that the DBMS interface is not included in the RADD/raddstar yet, it is not refered anymore in the following. The DBMS (Postgres) interface will be available as soon as possible, and– as Figure 7.6 shows –Web (HTML) forms that will make use of the interface can already be generated.

8.1.1 The RADD/raddstar Subsystem

As figured by the green frame and the boxes "Data Dictionary", "Behavior Specifications", and "Tuple Numbers" in Figure 8.1, the communication between the schema editor and the raddstar subsystem is realized by different file formats:

1. The *data dictionary file format* (".dd"), which stores all information necessary to maintain the graphical entity-relationship design. The raddstar system is usually invoked from the schema editor, such that it loads the data dictionary at startup.[1]
2. The *tuple numbers file format* (".tunums"), which declares tuple numbers (numbers of tuples) for the structures that are defined by the data dictionary. The tuple number files are generated and maintained by the raddstar.

[1]The other RADD components, the NLI, the semantics acquisition, and the strategy advisor, are also invoked from the RADD schema editor.

3. The CSL file format, which contains statements of the CSL database design specification language that was developed in this work. The generation and maintenance of these files is up to the RADD/raddstar as well.

Figure 8.2 shows the control flow and process architecture of the RADD/raddstar.

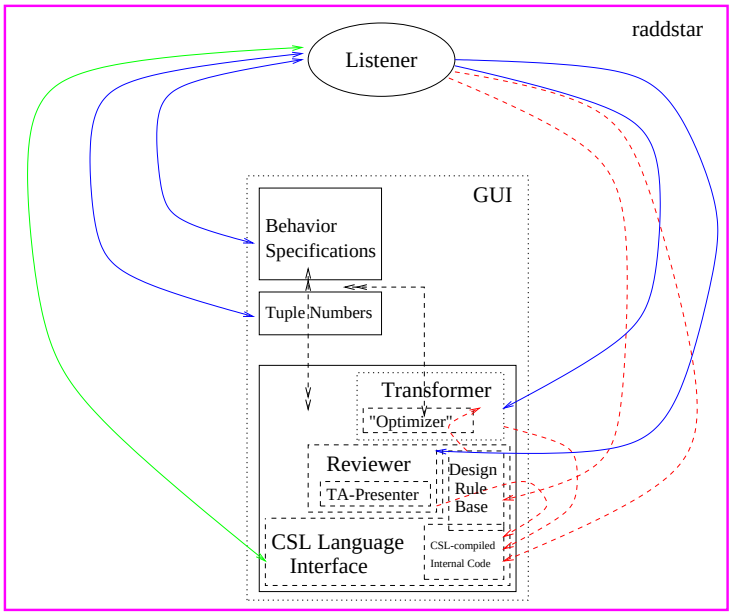


Figure 8.2: RADD/raddstar GUI Control Flow and Process Architecture.

The most important part of the RADD/raddstar system is a **Listener** that is controling the buttons of the GUI and the CSL shell, which is the RADD/raddstar command line interface and part of the RADD/raddstar GUI. All user actions– which are made either by entering CSL commands or else by clicking on the buttons of the GUI –are directed to the Listener firstly, and are then, after successful (or erroneous) interpretation, send to items of the GUI. The Listener takes care that no button action and CSL input is made during the review or optimization process, and it synchronizes the buttons upon themselves. For instance, the Listener invokes the update of the pulldown lists of the buttons labeled "Path(parent)", "Path(child)", and "Constraint", which are shown in Figure 6.8, after a new data dictionary is loaded into the RADD/raddstar.

In Figure 8.2 the green double-headed arrow between the box "CSL Language Interface" and the circle "Listener" indicates the control flow which is passed from the CSL shell to the Listener, and from the Listener to the CSL shell, on the other hand. The blue double-headed arrows between the boxes "Behavior Specifications" and "Tuple Numbers" and the circle "Listener" indicate the control flow between these interfaces and the Listener. The blue arrows from the Listener to the boxes "Transformer" and "Reviewer" indicate that the Listener invokes these processes (schema transformation and schema

optimization). The rules used for schema transformation and schema optimization, which are defined by means of the ".raddstar" startup file or by the CSL shell, are stored by the "Design Rule Base". The schema transformation and schema optimization write the results that they evaluate as "CSL-Compiled Internal Code" to the raddstar kernel, as does the Listener for the user's function and value defintions that it gets from the CSL shell.

Another architecture is

1. running the CSL shell (the "CSL Language Interface") in an X-terminal (xterm),
2. and running the graphical elements by a separate GUI.

This implementation has been shown to run more stable compared to the RADD/raddstar GUI, which has problems with "reviewing" large schemata, such as the extended Company schema which contains 28 entity, relationship, and cluster types. The latter architecture consisting of the "Listener", the "Listener GUI", and the "xterm" which is running the CSL shell, is shown in Figure 8.3.

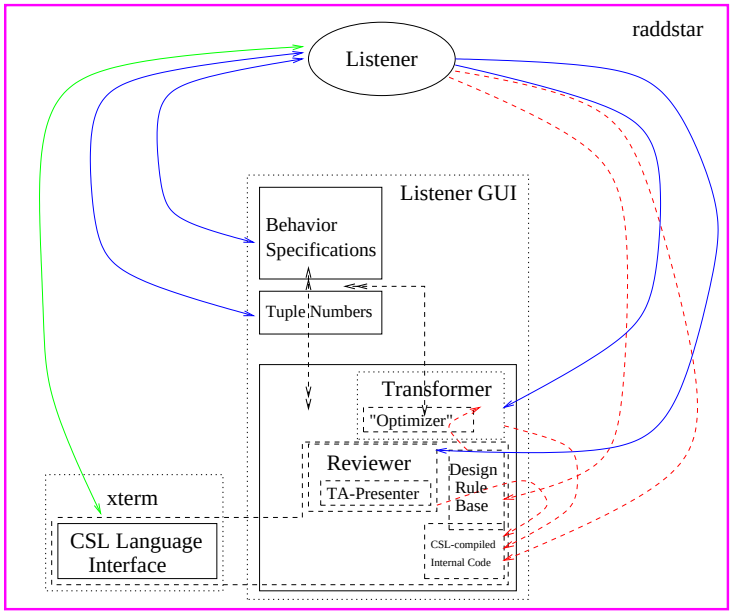


Figure 8.3: RADD/raddstar Listener GUI Control Flow and Process Architecture.

In addition to the buttons of the RADD/raddstar GUI, the RADD/raddstar Listener GUI has at the top a line which displays messages that are sent to the Listener. Since there is no other difference according the functionalty between the architecture shown in Figure 8.2 and the architecture shown in Figure 8.3, in the following we do not explicitely mention that difference of the GUI and the GUI/xterm. That is, in the following "CSL shell" and "CSL language interface" refer to both, the CSL shell running in the RADD/raddstar

GUI and the CSL shell running in the xterm. Accordingly, the references to the buttons of the RADD/raddstar GUI refer to both, the buttons of the RADD/raddstar GUI shown in Figure 6.8 and the buttons of the Listener GUI. The code shown in Appendix C.2.2 and Appendix C.2.3 gives an impression of the implementation of the Listener, the Listener GUI, and the CSL shell which is running in the xterm.

Although we had described the following commands already in Chapter 7, for the reader who wants to read the thesis quickly, here is a survey on them:

1. Besides loading the current schema of the schema editor at startup time, the database designer has the ability to load a new schema into raddstar by the *load data-dictionary* command of the CSL language (`load DD from "..."`). Whenever a new data dictionary is loaded into raddstar, maybe at startup time, maybe by explicitely entering this command, the tuple numbers and the CSL specifications are also loaded into the raddstar, if the according files exist. For instance, when raddstar is invoked from the schema editor with the data dictionary "Company.dd", it looks for "Company.tunums"– to get the tuple numbers –, and for "Company.csl"– to get the CSL specifications –and loads these files as well.
2. The conceptual schema, the internal schema[2], and the optimized (conceptual) schema can be exported in data dictionary format, such that they can subsequently be loaded into the schema editor. The schemata of the different types can also be exported into ascii files with SQL data definition language (DDL) statements. The exported SQL files contain the necessary code to create the tables (relations), foreign-keys (references), views, and triggers for the database.
3. The database designer can define additional constraints and maintenance rules for the graphical entity-relationship schema and invoke the schema fitness evaluation ("reviewing") and optimization. For this purpose, the RADD/raddstar graphical user interface provides a shell to invoke these processes and to enter CSL specifications.
4. At startup time, the raddstar looks firstly for an initialization file ".raddstar", from which it then loads its default transformation rules and behavior specifications. The transformation rules and behavior specifications in the ".raddstar" file, which is in the CSL format, override the rules and settings which are compiled into the system.[3]

An impression of the RADD/raddstar's graphical user interface was given by Figure 6.8.

[2]The internal database schema, that is derived from the conceptual schema to evaluate the operational behavior and the fitness of the operations.

[3]The current RADD/raddstar system does not anymore make use of precompiled schema transformation rules– like CoDO [Ste96] did. So, the initial transformation and optimization rules must (!) be loaded by means of the ".raddstar" file, from according CSL files, or must be entered on the interactive CSL shell. The CSL code for the definition of these rules is shown in Appendix B.

8.1.2 The Graphical User Interface of the RADD/raddstar

In the graphical user interface (see Figure 6.8), the CSL shell is placed on the right. Above of the CSL shell, on the right of the upper button line, the graphical user interface has three buttons (”Tuple Numbers”, ”Review Schema”, ”Optimize Schema”), which are shortcuts for some actions which can be invoked from the CSL shell:

- `enter tuple numbers` (invocation of the tuple number editor),
- `review conceptual schema`, and
- `optimize conceptual schema`.

The three buttons on the left of the upper button line are shortcuts for CSL commands as well, and are used for file maintenance (”Load DD”, ”Export DD”, ”Export SQL”). That are the CSL commands

- `load DD from "<filename>"`,
- `export DD for conceptual schema to "<filename>"`, and
- `export SQL for conceptual schema to "<filename>"`.

These buttons are related to the input field right of the ”Filename:” label, whose value is used as `<filename>` for these actions.

Behavior Specification Buttons. There is another menu (a matrix) below the ”Filename:” label and the input field right of the label, which is used for entering the behavior specifiations: the designer can so comfortably specify ”on insert {restrict,cascade,set null,set default}” and ”on delete {restrict,cascade,set null,set default}” options, for the parent and child structure (”Paths”) which are selected on the buttons labeled ”Path(parent)” and ”Path(child)”, and for the constraint on the button labeled ”Constraint”. The database designer can switch to another matrix with buttons for ”on update(child) {restrict,cascade,set null,set default}” and ”on update(parent) {restrict,cascade,set null, set default}” options, by clicking on the ”Ins/Del”-button, on the left upper corner of the matrix. The three buttons, ”Path(parent)”, ”Path(child)”, and ”Constraint”, have pulldowns which are listing the possible parent and child structures, or the constraints of the currently loaded schema. If the database designer does not select special parent and child structures and no special constraint, the system assumes that he makes the specification for the whole schema; that is, for all structures and constraints of the current schema *(general behavior specification)*. If he selects a parent or child structure and/or a special constraint, raddstar handles it as a *special behavior specification*.

Initially loaded CSL Files. The *global behavior specifications* should be predefined by the ".raddstar" file (or the `<filename>.csl` file), since otherwise raddstar's precompiled global behavior specifications (which are "on insert restrict", "on delete cascade", "on update(Parent) set null", and "on update(Child) cascade") are used. As Figure 6.8 shows, the current setting for the global behavior specifications is "on insert cascade" and "on delete cascade", such that the default option "on insert restrict" was overridden by either the ".raddstar" or the "Company.csl" file.

Transformation Types and Cost Models. Below the matrix for defining the behavior specifications is a button by which the database designer can select his prefered transformation type. Every transformation type is represented by the transformation rules for this type. The transformation rules are maintained by the CSL language and are initialized by the ".raddstar" (respectively `<filename>.csl`) file. Also, every transformation type is related to a set of different cost models, such as `ISAM`, `Traditional Btree`, `Dense Btree`, or `Extensible Hash`. One of these is the default cost model for the transformation type, which is then initially set for that transformation type. The cost model that is actually set, is used together with the transformed conceptual schema and the set of behavior specifications ("on insert cascade", etc.) to evaluate the contents and the complexities of the select, insert, delete, and update operations, for the structures of the conceptual schema.

From Figure 6.8 we recognize, that the current setting of the transformation type is `Relational`, which is shown right of the button, and the current setting of the cost model is `Traditional Btree`, which is indicated by the button labeled "Btree". For each cost model the user has the ability to configure its "Balancing Parameters", which give the evaluated cost term or cost term which is to be evaluated, the less or the more weight, such that the behavior estimator of the raddstar ("Reviewer", Figure 8.1) evaluates different complexities for select, insert, delete, and update operations.

8.2 Specifying Additional Requirements

After the user has finished the graphical design, the schema that he designed can be annotated with additional requirements. The additional requirements can be

- *tuple numbers* which are specified according the entity, relationship, and cluster types of the conceptual schema ("classes"),
- *behavior specification* which are defined for the integrity constraints, and
- *database functions* which are annotated to the schema, or, to the classes– in a fashion of class or member functions.

8.2.1 Tuple Numbers

The tuple numbers are used for the evaluation of the complexities of the transactions which are generated for the conceptual schema. The select, insert, delete, and update operations, and the operations which are possibly triggered by them have other complexities depending on the tuple numbers of the database relations, e.g. if the tuple number is higher then an insert operation is more expensive. Also, if an insert operation can trigger another insert operation or requires that another insert operation is executed previously, then the cost of the first insert operation includes (a part of) the cost of the second insert operation.[4]

Assume the conceptual RADD (HERM) schema shown in Figure 3.16 has been loaded in the raddstar. The data dictionary ("Company28.dd") contains 28 entity, relationship, and cluster types. Also, a tuple specification file ("Company28.tunums") has been loaded in the raddstar, that defines tuple numbers for 8 types of the graphical design. Then, the tuple numbers can be adjusted and specified using the tuple number dialogue, which can be invoked either by entering the *enter tuple numbers* command or by pressing the "Tuple Numbers" button. The dialogue is shown in Figure 8.4.

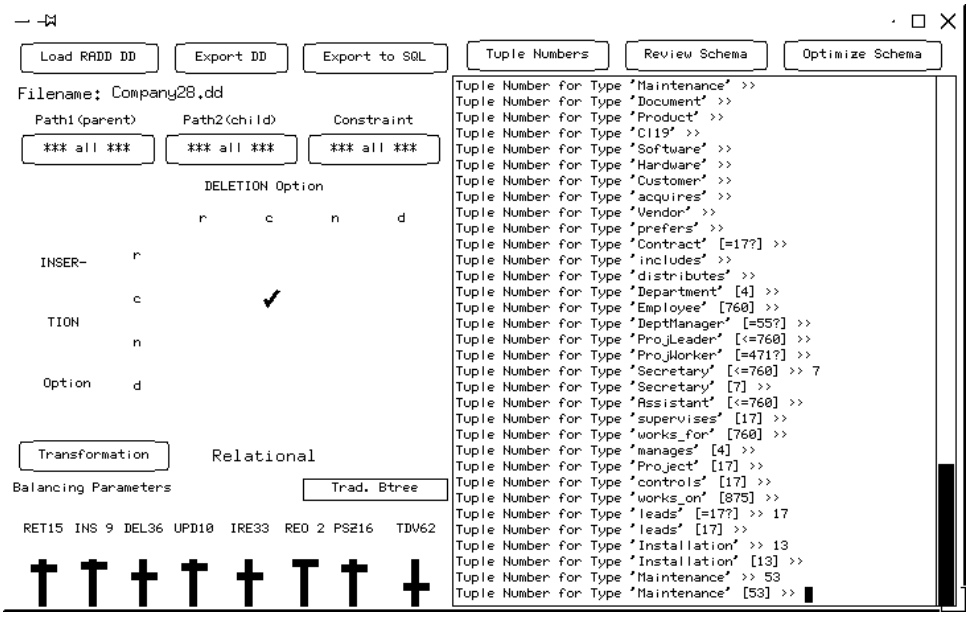


Figure 8.4: Specifying the Tuple Numbers for the Classes of the Schema.

The tuple number dialogue shows how the database designer defines and fixes the tuple numbers for some classes, for which no tuple numbers where defined in the tuple number specification file, such that the system tries to infer them from the given tuple numbers of connected structures and the cardinality constraints.

[4]For details of the cost model and the transaction evaluation refer to Chapter 5.

Representation by the data model. According the RADD* data model that we defined in Section 6.3, the tuple numbers are internally represented by the 'TupleNumber' slots of the Structs (RADD/HERM entity and relationship types) and the Unions (RADD/HERM cluster types).

8.2.2 Behavior Specifications

Figure 8.5 shows how behavior specifications are added to the integrity constraints of the graphical design.

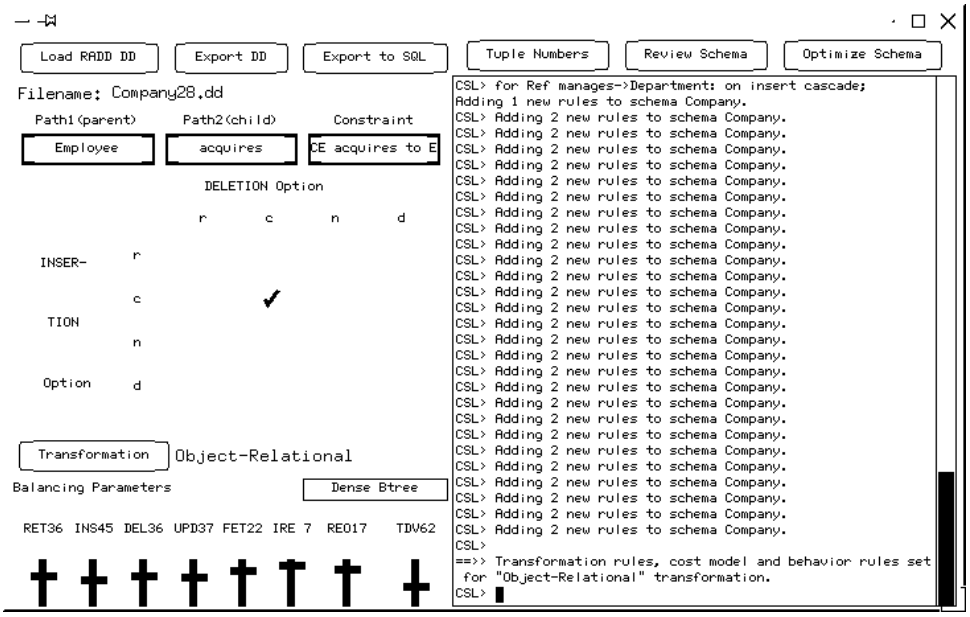


Figure 8.5: Specifying Behavior for the Graphical Schema.

The behavior specifications are used for the evaluation of the conceptual schema transactions, and have also influence to the schema transformation processes. Presuppose, the supervises class has a reference to the Project class, and the according cardinality constraint is (1,1) ($card(supervises, Project) = (m, n)$). Furthermore, assume a transformation rule says to group (collapse) all structures which are 1:1. Then, if a behavior specification *on-insert-cascade* is given for the reference or for the cardinality constraint, Project and supervises are not grouped by the transformation. On the internal schema that is used to evaluate the transactions, the *on-insert-cascade* would otherwise not appear in form of a triggering action– since the two structures of the conceptual schema were grouped to one internal structure. This would us not enable to present the triggering action for which he specified the *on-insert-cascade* to the database designer.

Representation by the data model. According the RADD* data model that we defined in Section 6.3, the behavior specifications are internally represented by the 'be-

havior option/specification set' slots of the references of Structs (RADD/HERM entity and relationship types) and Unions (RADD/HERM cluster types).

8.2.3 Database Functions

Database functions are specified by means of the `add class` feature of the specification language. An example is shown in Figure 6.8.
Whenever functions are added to the classes of the graphical design, then (1.) the identifiers are checked for their binding (in the `add class` example in Figure 6.8, `this` is bound to (a member record of) the Employee class), (2.) the functional declaration is type checked and compiled,[5] and (3.) the function is added to the classes internal representation.

Representation by the data model. According the RADD* data model that we defined in Section 6.3, the database functions are internally represented by the 'application modules' slots of the Structs (RADD/HERM entity and relationship types) and the Unions (RADD/HERM cluster types).

8.3 Schema Reviewing and Optimization

After the user has specified additional requirements– which is an optional feature, he needs not to do that –the conceptual schema must be reviewed to gather criteria for schema bottlenecks and optimization.

8.3.1 Schema Reviewing

According the adjusted tuple numbers, the behavior specifications, and the functions that are added to the classes, the behavior estimator of the RADD/raddstar (Reviewer) presents the cost matrix in Figure 8.6 to the user.

From the cost matrix, it is easy to recognize that the database user may be confronted with unreliable response times when performing some operations. So, under certain circumstances

- the insert operation for the entity types *Assistant*, *Department*, *DeptManager*, *Employee*, *Product*, *ProjLeader*, *ProjWorker*, *Secretary*, the relationship types *acquires*, *includes*, *works_for*, and *works_on*, and the cluster type *Cl*19,

[5] The CSL functional specification language, its type inference and compilation, are described in the second Section of Chapter 6.

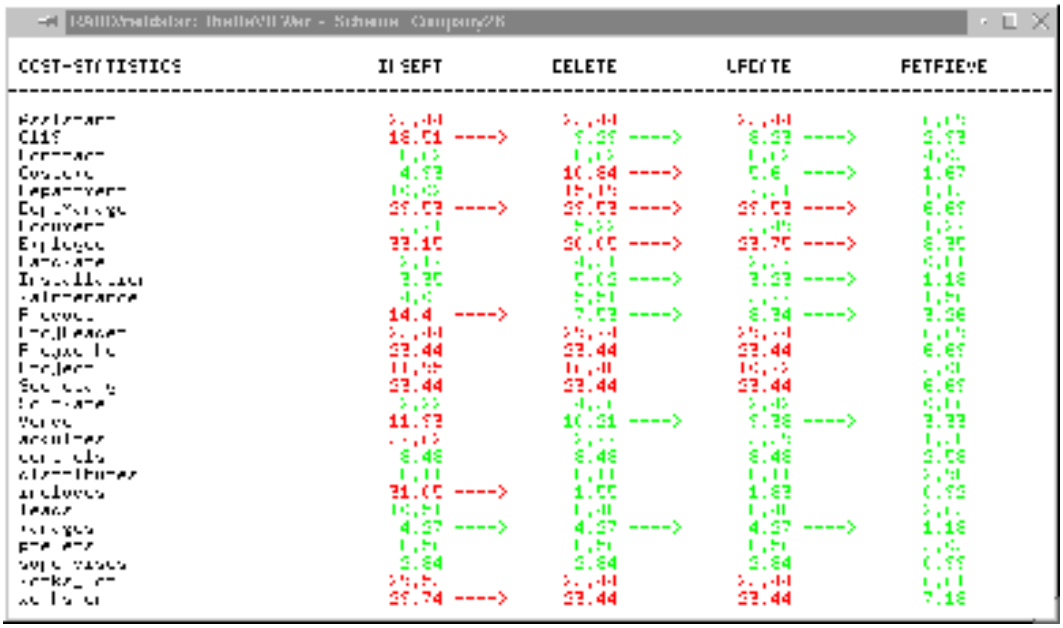


Figure 8.6: Matrix presenting the Transactions of the Company Schema.

- the delete operation for *Assistant*, *Customer*, *Department*, *DeptManager*, *Document*, *Employee*, *Hardware*, *Installation*, *Maintenance*, *Product*, *ProjLeader*, *ProjWorker*, *Project*, *Secretary*, *Software*, *Vendor*, *acquires*, *includes*, *works_for*, *works_on*, and *Cl*19, and
- the update operation for *Assistant*, *Department*, *DeptManager*, *Employee*, *ProjLeader*, *ProjWorker*, *Secretary*, *works_on*, and *works_for*

may be possible bottlenecks.

Discussion. Whether this operational behavior is really a bottleneck of the schema depends upon the frequency and priority the mentioned operations are required. More specific:

- if insertion is an often required operation for the entity types, then the insert operation creates a crucial bottleneck;
- if it is necessary to delete frequently records of the entity and relationship types, then the delete operation has higher complexity;
- the update operation for the entity types *Assistant*, *Department*, *DeptManager*, *Employee*, *ProjLeader*, *ProjWorker*, and *Secretary*, and for the relationship types *works_for* and *works_on* could be considered as relatively complex.

Therefore, we can derive that

1. if we need frequently the above mentioned operations then we must recommend that
   - the *manages* type is a subtype of *works_for* identifying the special role of department managers
   - the types *DeptManager*, *ProjLeader*, and *ProjWorker* are marking special roles of the *Employee* and seem to be dummy types
     (therefore, we could reason whether these types could alternatively be represented as an attribute– representing that special role)
   - the relationship types *controls* and *leads* are both associated 1:1 to the *Project* entity type (this way, it were possible to group these types to one new relationship type);
2. if the entity occurence sets are not changed frequently then above discussed relationship types should be grouped in the case that their update operations get a very high frequency and priority
3. the insert, delete, and update operations for *works_for* and *works_on* should be considered as important in frequency and priority; a requirement that confirms this high priority can be that updating an *Employee* to work on a new or another *Project* is an action that is frequently executed by the database user interfaces; therefore, we can ask for criteria identifying that this type should be used to create a separate data file.

Specific restructuring suggestions and how the suggestions are specified by the database designer are discussed below. Further rules for optimization can be developed in accordance to tuning techniques of the chosen DBMS.

8.3.2 Bottleneck Specification and Schema Optimization

In figure 8.4, we showed how the user maintains the tuple numbers. We also showed in Figure 8.5, how he can graphically introduce the behavior specifications. We discuss now how he specifies further properties of the data profile of the conceptual schema. These additional capabilities of conceptual database specification using RADD/raddstar consist of

- specifying integrity maintaining rules,
- informally specifying which operations are frequently required and have high priority, and which are not frequently required and have no high priority, and
- defining primary storage organizations and indices for reasons of faster query processing.

8.3.2.1 Integrity Maintaining Rules

The subtransaction sequence of the insert$_{works_on}$ operation in Figure 7.4 contains no operations which retrieve and modify the *Employee* set, although insert into the *works_on* set normally either *requires* a previous insert of the associated *Employee*, or else *triggers* an insert into the *Employee* set. This is not the case because the *works_on* structure has been grouped into the *Employee* structure– by the transformation to the internal schema. But maybe, the user wants to see how the transaction insert$_{works_on}$ works. For example, it is possible that he specifies

```
for Ref works_on->Employee: on insert cascade;
```

or

```
for Ref works_on->Employee: on update child cascade;
```

such that the according grouping is not done, since, for reason of the user's cascade rule, the RADD/raddstar derives that the user wants to see the content of the transaction.

This way, the user has influence to the transformation process, but he has also the opportunity to explicitely advise the system to do or to omit transformations, e.g.:

```
CSL> add objectrelational transformation rule:
      do not group (Employee,works_on);
Adding new objectrelational transformation rule as "o3".
```

Given such a rule, the raddstar does not group (Employee,works_on) when deriving the internal schema. On the other hand, if at a later date, the user wants to drop a behavior rule (or transformation rule) again, he can use the 'drop' 'rule' command, e.g.

```
drop rule ( for Ref works_on->Employee: on insert cascade );
```

and

```
drop rule "o3";
```

8.3.2.2 Operation Frequencies and Priorities

The user can informally specify that operations are frequently required, and that they have high priority:

```
CSL> operation (update,Employee) is of_high_priority;
property added.
CSL> operation (update,Employee) is frequently_required;
property added.
CSL> operation (delete,Project) is not frequently_required;
property added.
```

If an operation is frequently required or has high priority, then the threshold where it is assumed to be a bottleneck is lower. E.g., an insert operation with cost 9.8 is considered a bottleneck and an operation with cost 9.3 is considered not to be a bottleneck. But an insert operation with cost 8.0 which is "frequently_required" is considered a bottleneck as is an operation with cost 6.6 which is "frequently_required" and "of_high_priority".

8.3.3 Optimized Schema

Figure 8.7 shows what the optimized conceptual schema that is evaluated by the RADD/-raddstar system, looks like. The items of the data profile (tuple numbers, integrity maintaining rules, operation properties, transformation rule, optimization rule) that we mentioned in the Section had influence in the optimization of the schema.

The schema is now reviewed again. The applications used for the reviewing and optimization, and the cost matrix for the "optimized" schema are shown in Figure 8.9, at the end of the Section.

8.3.3.1 Adding Physical Data Access Methods

As we see from the cost matrix in the lower right corner in Figure 8.9 some operations are now more expensive or not much less expensive. E.g., the $\text{delete}_{Employee}$ or the insert_{works_on} operation. This can be reasoned by the new schema which has a lower number of structures, such that a particular structure may now have more attributes, and by the fact that *works_on* is now no longer a part of the *Employee* set of the internal schema. But, in the RADD/raddstar the user can assign storage options to the structures of the conceptual schema which make query processing and the associated update operations (insert, delete, update) quicker.

Assume, most times the *Company* has relatively less personel according the allocated projects. Then, for a newly started project it may be necessary to get personel from other projects, that are still available according the hours they are scheduled on projects. This way, it may be wishful to attach special storage organizations to the structures (primary-key based organizations) and to attribute combinations (secondary indices), which are frequently required, e.g. in join operations. The following dialogue shows how the user 'modifies' the storage organizations.[6]

```
CSL> modify class Employee{Salary} to Btree;
==>> adding secondary index of type "Btree" to Employee{Salary}.
```

[6]Normally, a structure can have only one clustering index, such as "Btree" (traditional, sparse and clustering Btrees). But remember that the user has chosen the "DBtree" model (dense non-clustering Btrees) as his prefered primary storage organization.

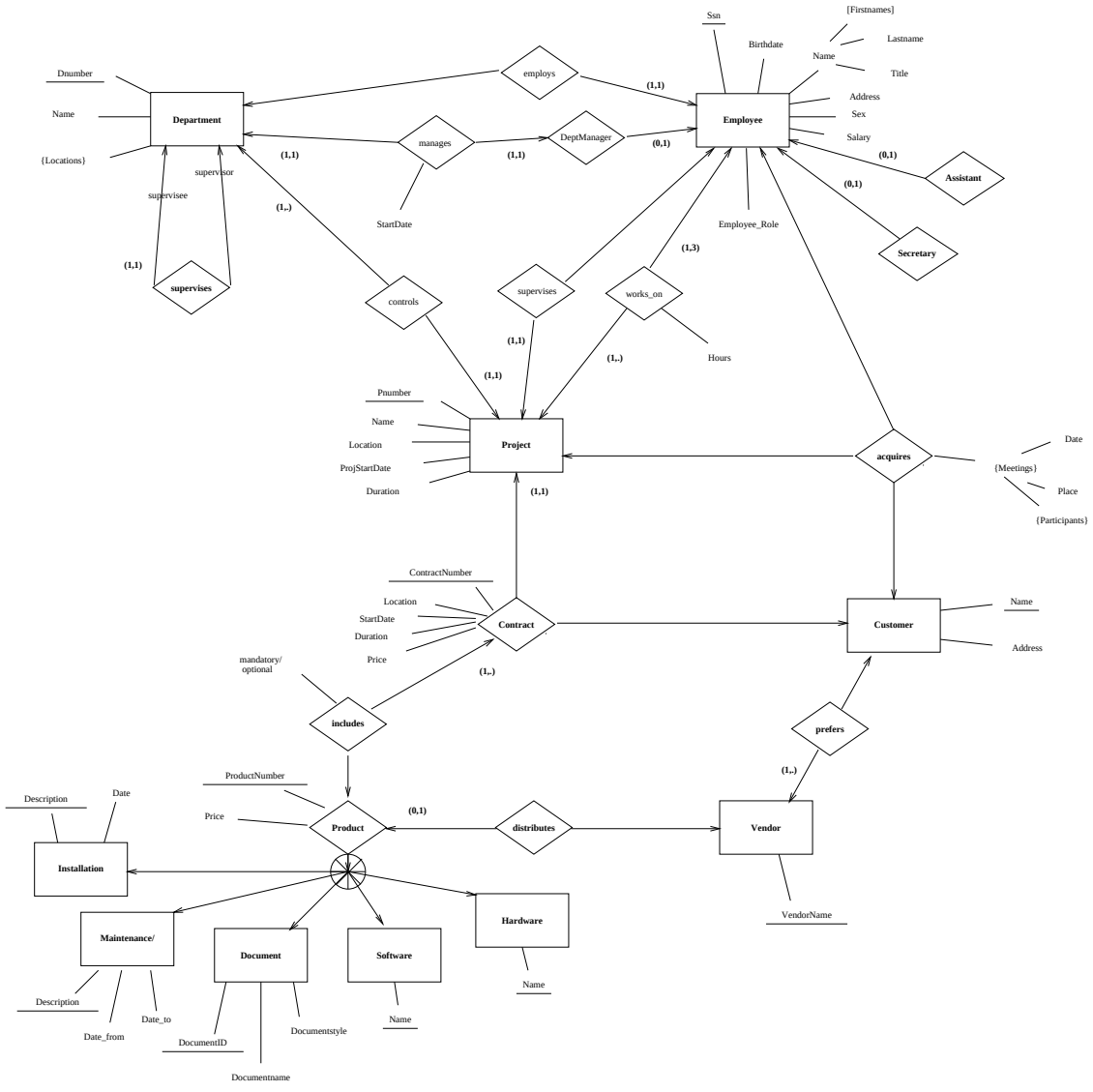


Figure 8.7: Optimized Company Schema.

```
CSL> EHash;
it : storage_organization = EHash;
CSL> modify class works_on{Employee} to it;
==>> adding secondary index of type "EHash" to works_on{Employee}.
CSL> modify class works_on{hours} to Isam;
==>> adding secondary index of type "Isam" to works_on{hours}.
CSL> modify class works_on{Project} to Btree;
==>> adding secondary index of type "Btree" to works_on{Project}.
```

The impacts of these attachments of storage organizations are as follows

1. the command which adds a (clustering) Btree index to `Employee{Salary}` allows that range queries on the *Salary* are executed fast; that is, if a query like

   ```
   select e.* from Employee e where Salary < 60000
   ```

   is an often required query, then this index is appropriate to make the query run fast.
2. the command which adds an EHash index to `works_on{Employee}` allows that point queries on the *Employee-Project* association are quickly executed. This is necessary if queries like the following have higher priority

   ```
   select p.Name,e.*,w.Hours from Project p,works_on w,Employee e
    where p.ProjNumber = w.Project and w.Employee = e.Ssn
    order by p.Name,e.Name.Lastname
   ```

3. the command which adds an Isam index to `works_on{hours}` is appropriate if the total time which the *Employees* work on a particular *Project* is often computed

   ```
   select p.ProjNumber,p.Name,sum(w.Hours) from Project p,works_on w
    where p.ProjNumber = w.Project
    group by p.ProjNumber,p.Name
   ```

4. the command which adds a (clustering) Btree index to `works_on{Project}` provides that the number of *Employees* which are working on a *Project* can be computed efficiently

   ```
   select p.ProjNumber,p.Name,count(e.*) from Project p,works_on w,Employee e
    where p.ProjNumber = w.Project and w.Employee = e.Ssn
    group by p.ProjNumber,p.Name
   ```

The modify statements described here are only a few examples for optimizing (query) processing on the Company Schema. The new cost evaluation which is shown below has been generated adding some more secondary indices, and modifying the primary indices such that the storage allocation (on inserts) and the storage reorganization (on inserts and updates) work faster. For more examples and a detailed discussion how to use indices and primary storage organizations, the reader may refer to [Wie87], [KS91], and [Sha92].

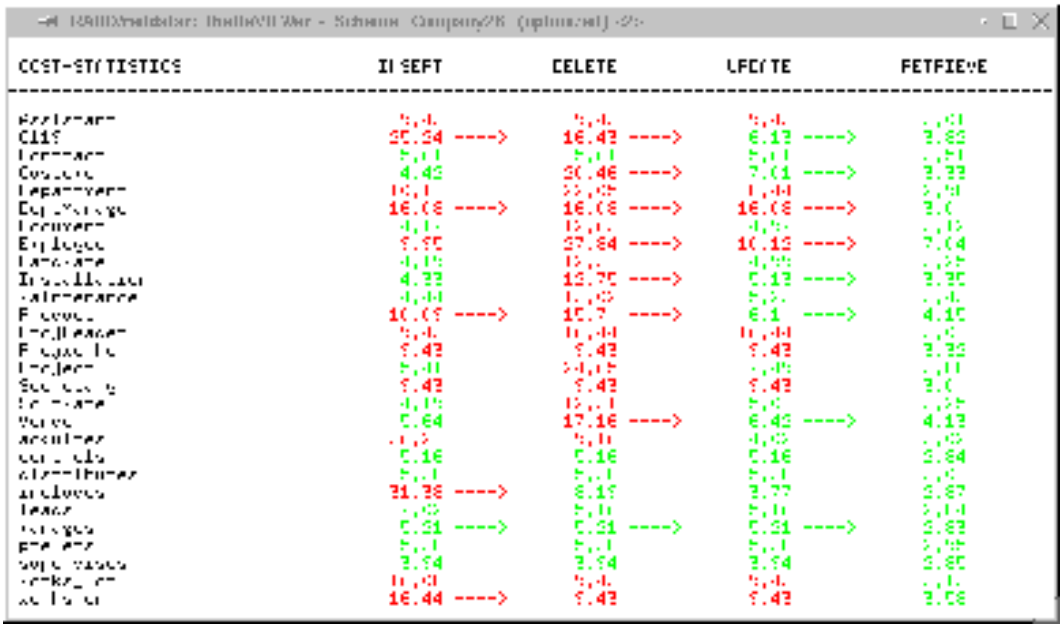


Figure 8.8: Transaction Costs of the Optimized Schema, after Adding the Indices.

As mentioned in Section 6.3, the storage options are annotated to the structures, and are considered in the cost evaluation of the associated operations.

The cost matrix in Figure 8.8 shows how some costs are decreasing considering the indices and primary storage organizations, which have been specified for the the optimized schema. Further, the user can experiment to improve the performance by using different balancing parameters, *block size, bucket size, fanout, etc.* Their relative size can be configured using the RADD/raddstar GUI and the RADD/raddstar Listener GUI.

Exporting the optimized schema. Finally, the optimized schema can be exported to a *Data Dictionary* for the RADD schema editor, and the internal schema can be exported to SQL create table und index commands.

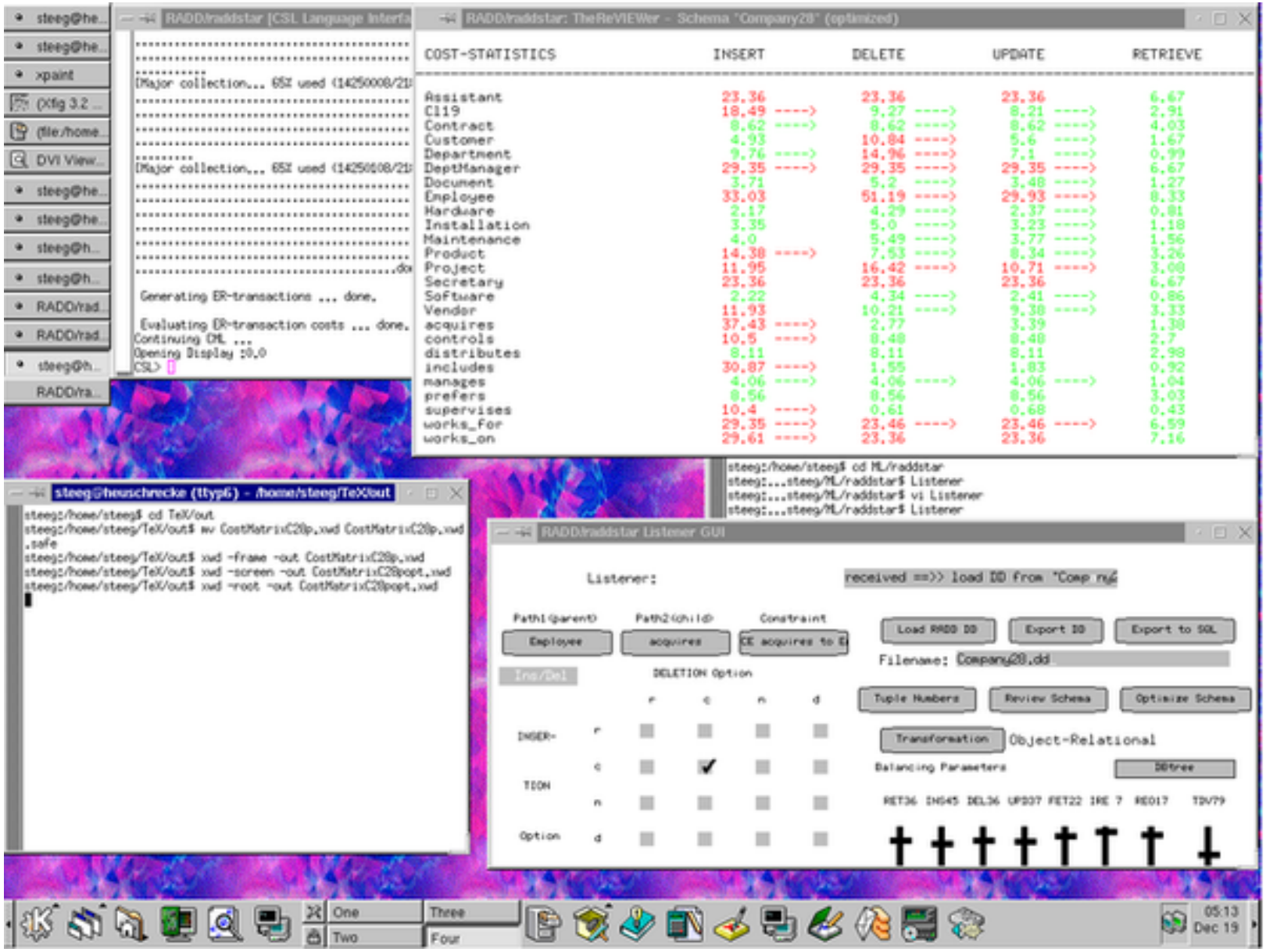


Figure 8.9: Matrix presenting the Transactions of the Optimized Company Schema.

Chapter 9

Conclusions

The tool presented here supports database design according to practical design issues, which include the performance of batch transactions and the reponse time of user-interface actions. Practical issues are also the consistent behavior of the database applications, and performance is not only reaction time of selects, inserts, deletes and updates, or, reaction time of database access in certain user menus, but also, that data can be quickly retrieved, generated, modified, or deleted.

The latter aspects create requirements of the user-interfaces' easy-usability, where it is often desired to introduce related data by a single menu action, but not on basis of many hierachical dependent menu paths which must be traversed by the user. This can require to omit normalization or to denormalize database schemata, such that simpler menu paths can be provided. This, in turn, can cause problems of inconsistent or blocking transactions.

The RADD workbench and the RADD/raddstar system consider these criteria of (internal) database design, and are supporting the design of correct and application-reliable conceptual schemata. This way, database restructuring or redesigning requirements once performance and/or consistency problems are detected, can be farreaching avoided.

Storing the History of the Database Design and Transformation Process. A feature which is often omitted in other database design approaches, that has been realized in the RADD/raddstar, is to preserve, store, maintain, and reuse historical aspects during the database design and transformation process. Additional requirements can be specified after the graphical database design, by means of the CSL language and the graphical user interface of the RADD/raddstar. After these actions, RADD/raddstar evaluates the conceptual schema on the basis of a rule-driven mechanism to support

- Schema Transformation,
- Operational Cost Evaluation, and
- Bottleneck Detection and Visualization.

The results of the RADD/raddstar's evaluation can then be used to optimize the conceptual schema– with the help of the database designer who can agree or deny the system-detected bottlenecks.

RADD/raddstar is implemented on top of the Standard ML of New-Jersey functional programming system (SML/NJ 0.93). Although it provides with CSL a specification language that has a declarative flavor, its compilation and evaluation kernel is a pure functional machine which is based on the λ-calculus.

The RADD/raddstar system can read specifications for graphical database designs– more specific, the *data dictionaries* of the Cottbus University toolbox for *Rapid Application and Database Development* (RADD). The system is able to export internally evaluated, optimized conceptual data schemata in the RADD language. The RADD* data model presented in Chapter 6 states a superset of different database views, i.e. conceptual schemata (of ER models, object models, ORM, etc.) and hierarchical, network, relational, object-relational, and object-oriented databases. The behavior specification interface (CSL) permits the user to introduce behavior rules for the conceptual schema, which are later used in the database schema definition. Terms like "on insert set default" or "on delete cascade" are examples for such behavior rules.

Acknowledgements. I primarily want to thank Prof. Thalheim of Cottbus Technical University who supported this work. I also like to thank the colleagues and a former colleague of IABG, who carefully read the thesis and helped to improve it.

Appendix

Appendix A

Implementation of a Type-Checking Mini ML Compiler

This Chapter presents the code of the Standard ML like functional language that we have discussed in Chapter 6.2.2. The code of this Chapter also represents the basic evaluation and compilation kernel of the RADD/raddstar.

A.1 Basic Types of the Mini ML Compiler

```
(*
 * RSsml.sml
 *
 * Raddstar Structured Meta Language
 * Expression Compiler -- Basic Structures and their Constructors/Destructors
 *
 * Copyright Martin Steeg, 1996 - 1999
 *)

signature RSSML =
 sig

  exception SUnbound of string
  exception TypingBug of string

  val in_l : ''a * ''a list -> bool
  val posl : ''a list * ''a -> int
  val subl : ''a list -> ''a list -> ''a list
  val make_ulist : ''a list -> ''a list

  val is_rssml_op : string -> bool
  val prio_rssml_op : string -> int

  type ident

  exception UNcaught of ident

  val IT : ident
  val FIX : ident

  val make_ident : string -> ident
  val stringof_ident : ident -> string

  val psln : unit -> unit
  val psln2 : unit -> unit
  val say : string -> unit

  type vartype
```

```
  type varenv_t

  datatype stmt = FUNDEF of ident * (value * value) list
                | VALDEF of ident * value
                | NULLSTMT
                | QUIT
      and value = VB of bool
                | VI of int
                | VS of string
                | VPARMS of value list
                | VVAR of varenv_t ref * int
                | VAPP of value * value
                | VBRANCH of value * value * value
                | VFUN of (value -> value) * vtype
                | VFNDEF of (value * value) list
                | VID of ident
                | VALLIST of value list
                | VOP of string
                | VNULL
      and vtype = bool_t
                | int_t
                | string_t
                | parms_t of vtype list
                | fun_t of vtype * vtype
                | typevar of vartype
                | noninit_t
    and vtypesc = Forall of vtype list * vartype

  val varsoftype : vtype -> vartype list
  val is_fully_initialized_type : vtypesc list * vtype -> bool
  val unknownsoftype : vtypesc list * vtype -> vartype list
  val unknownsofenv : vtypesc list -> vartype list

  val make_typesc : varenv_t -> vtypesc list
  val reset_vartypes : unit -> unit
  val get_current_env : unit -> vtypesc list
  val set_current_env : vtypesc list -> unit
  val make_new_typevar : vtype ref list ref * vtypesc list ref -> vtype
  val make_typevar : unit -> vtype

  val is_initialized_type : vtype -> bool

  val stringof_value : value -> string

  val descrtype : vtype -> string
  val get_vartype : vartype -> vtype
  val set_vartype : vartype -> vtype -> unit
  val findtypeenv : vtype -> vtype
  val dispatch : varenv_t ref * string -> value
  val vtypeofv : value -> vtype
  val typeofvar : varenv_t ref * int -> vtype
  val vtcomp : vtype * vtype -> bool
  val occursintype : vartype -> vtype -> bool

  val make_venv : (string * value) list -> varenv_t
  val make_vvar : varenv_t ref * string -> value
  val vnormalize : varenv_t ref -> value -> value
  val vnormalizel : varenv_t ref -> value -> value

  val local_venv : varenv_t ref

 end

structure RSsml : RSSML =
 struct

  exception SUnbound of string
  exception TypingBug of string

  fun in_l (e,[]) = false
    | in_l (e,a::l) = if e=a then true else in_l(e,l)

  infix 9 in_l

  fun posl (l,e) = posl'(0,l,e)
  and posl' (_,[],_) = ~1
```

```
  | posl' (n,a::l,e) = if a = e then n else posl'(n+1,l,e)

fun sube [] a = []
  | sube (b::bs) a = if a = b then bs else b::(sube bs a)
and subl l [] = l
  | subl l (b::bs) = subl (sube l b) bs

fun make_ulist [] = []
  | make_ulist (a::l) =
     if a in_l l then make_ulist l else a::make_ulist l

(* search a value in a field of a list *)
fun findlist value f g def [] = def
  | findlist value f g def (b::bs) =
     if value = (f b) then g b else findlist value f g def bs

(* select the values of the list satisfying condition described by f *)
fun filterlist [] _ = []
  | filterlist (a::l) f =
     if f a then a::filterlist l f else filterlist l f

val rssml_op_tab =
     ref [ ("+",6), ("-",6), ("*",7), ("/",7),
           ("=",5), ("<",5), ("<=",5), (">",5), (">=",5), ("<>",5),
           (":=",3) ]

fun is_rssml_op s = s in_l (map (fn (a,_) => a) (!rssml_op_tab))

fun prio_rssml_op s =
     if is_rssml_op s
      then findlist s (fn (a,_) => a) (fn (_,b) => b) ~1 (!rssml_op_tab)
      else raise(TypingBug(s^" is no infix operator"))

fun make_rssml_op (s,p) =
     rssml_op_tab := (filterlist (!rssml_op_tab) (fn (a,_) => a<>s)) @ [(s,p)]

datatype ident = IDENT of int

exception UNcaught of ident

val init_ident_env =
              [ "it",
                 "not",
                 "if", "then", "else", "fi",
                 "true", "false",
                 "fun", "val", "fn",
                 "fix"
               ]

val IT = IDENT 0
val FIX = IDENT((length init_ident_env)-1)

type ident_entry = {b: bool ref, i: string}

type ident_entry_list = ident_entry list

val gident_env = map (fn id => {b=ref true,i=id}) init_ident_env

val lident_env = ref gident_env

fun position_ienv s =
     let val ret = ref ~1
         val pos = ref 0
         val e = ref (!lident_env)
     in
      while !ret = ~1 andalso (!e) <> [] do
       case hd(!e) of
          {b=ref true,i=id} =>
            if s=id then ret := !pos else (e := tl(!e); pos := !pos+1)
        | _ => (e := tl(!e); pos := !pos+1);
      if !ret = ~1
       then raise(SUnbound s)
       else !ret
     end

fun append_ienv s =
     case (findlist s (fn {b=_,i=id} => id) (fn {b=b,i=_} => (b,true)) (ref false,false) (!lident_env)) of
```

```
        (ref true,_) => ()
      | (b,true) => b := true
      | _ => lident_env := (!lident_env) @ [{b=ref true,i=s}]

fun stringof_ienv env n =
     (case nth(!env,n) of
         {b=ref true,i=id} => id
       | _ => raise(SUnbound ("*** No valid id on position "^(makestring n)^" of ident_env ***")))
      handle Nth =>
       raise(SUnbound ("*** No id on position "^(makestring n)^" of ident_env ***"))

fun make_ident id = (append_ienv id; IDENT(position_ienv id))

fun stringof_ident (IDENT i) = stringof_ienv lident_env i

(* some definitions for the Scanner and Parser *)
val psln = fn() => output(std_out,"RSsml> ")
val psln2 = fn() => output(std_out,"> ")
val say = fn s => output(std_out,s)

datatype stmt = FUNDEF of ident * (value * value) list
              | VALDEF of ident * value
              | NULLSTMT
              | QUIT
    and value = VB of bool
              | VI of int
              | VS of string
              | VPARMS of value list
              | VVAR of (string * (value ref * vtype ref)) list ref * int
              | VAPP of value * value
              | VBRANCH of value * value * value
              | VFUN of (value -> value) * vtype
              | VFNDEF of (value * value) list
              | VID of ident
              | VALLIST of value list
              | VOP of string
              | VNULL
    and vtype = bool_t
              | int_t
              | string_t
              | parms_t of vtype list
              | fun_t of vtype * vtype
              | typevar of vtype ref list ref * int
              | noninit_t
  and vtypesc = Forall of vtype list * (vtype ref list ref * int)

type vartype = vtype ref list ref * int


(* val varsoftype : vtype -> vartype list *)
fun varsoftype typ =
     let fun vars vs (parms_t l) =
              fold (fn (a,b) => b@a) (map (vars []) l) vs
           | vars vs (fun_t(t1,t2)) = vs@(vars [] t1)@(vars [] t2)
           | vars vs (tv as typevar(e,p)) = vs@[(e,p)]@(vars [] (!(nth(!e,p))))
           | vars vs _ = vs
     in
      make_ulist(vars [] typ)
     end

(* val is_fully_initialized_type : vtypesc list * vtype -> bool *)
fun is_fully_initialized_type (_,noninit_t) = false
  | is_fully_initialized_type (sce,parms_t l) =
     let val ret = ref true in
      app (fn t => ret := ((!ret) andalso is_fully_initialized_type (sce,t))) l;
      !ret
     end
  | is_fully_initialized_type (sce,fun_t(t1,t2)) =
     is_fully_initialized_type (sce,t1) andalso is_fully_initialized_type (sce,t2)
  | is_fully_initialized_type (sce,typevar(e,p)) =
     let fun isfivt [] = false
           | isfivt (Forall(rl,t)::l) =
              if t=(e,p) then
               case rl of a::_ => is_fully_initialized_type(sce,a) | _ => false
              else isfivt l
     in
      isfivt sce
```

```
      end
  | is_fully_initialized_type _ = true

(* val unknownsoftype : vtypesc list * vtype -> vartype list *)
fun unknownsoftype (sce,typ) =
     fold (fn (vt,l) => if is_fully_initialized_type(sce,typevar vt) then l else vt::l) (varsoftype typ) []

(* val unknownsofenv : vtypesc list -> vartype list *)
and unknownsofenv sce =
     make_ulist(fold (fn (Forall(_,vt),e) => (unknownsoftype(sce,typevar vt))@e) sce [])


(* varenv_t defines the type of the local var env *)

type varenv_t = (string * (value ref * vtype ref)) list

(*
 * The local var env
 *  that will be initialized at the end
 *)

val local_venv = ref([] : varenv_t)


local

 (* (!type_env) holds the vartypes of the current expression *)
 val type_env = ref(ref([] : vtype ref list))
 (* (!typing_env) associates the vartypes with their bindings *)
 val typing_env = ref([] : vtypesc list)

in

 val make_typesc =
      fn venv =>
       fold (fn (a,b) => (map (fn (e,p) => Forall([],(e,p))) a)@b)
        (map (fn (_,(_,t)) => varsoftype(!t)) venv) []

 val reset_vartypes = fn () =>
      (type_env := ref[];
       typing_env := make_typesc(!local_venv))

 val get_current_env = fn () => !typing_env
 val set_current_env = fn sc => typing_env := sc

 val make_new_typevar =
      fn (typenv,env) =>
           (typenv := (!typenv)@[(ref noninit_t)];
            let val (e,p) = (typenv,length(!typenv)-1) in
             env := (!env)@[(Forall([],(e,p)))];
             typevar(e,p)
            end)

 val make_typevar = fn () => make_new_typevar(!type_env,typing_env)

end (* local *)

fun is_initialized_type noninit_t = false
  | is_initialized_type (typevar _) = false
  | is_initialized_type _ = true

fun stringof_value v = stringof_value' v
and stringof_value' (VB true) = "true"
  | stringof_value' (VB false) = "false"
  | stringof_value' (VI i) = makestring i
  | stringof_value' (VS s) = "\""^s^"\""
  | stringof_value' (VPARMS l) =
     let val r = ref"" in
      app (fn v => r := (!r)^(if !r = "" then "" else ",")^(stringof_value v)) l;
      "("^(!r)^")"
     end
  | stringof_value' (VVAR(e,p)) = (case nth(!e,p) of (n,_) => "Var(\""^n^"\")")
  | stringof_value' (VAPP(v1,v2)) = "("^(stringof_value v1)^")"^(stringof_value v2)
  | stringof_value' (VBRANCH(p,v1,v2)) = "if "^(stringof_value p)^" then "^(stringof_value v1)^
                                         " else "^(stringof_value v2)^" fi"
  | stringof_value' (VFUN _) = "<function>"
  | stringof_value' (VFNDEF l) = "fn "^(let val r = ref "" in
```

```
                                        app (
                                         fn (e,v) =>
                                          r := (!r)^(if !r = "" then ""
                                                                else "|")^
                                                (stringof_value e)^"=>"^(stringof_value v)
                                        ) l; !r end)
  | stringof_value' (VID i) = stringof_ident i
  | stringof_value' (VALLIST l) = "("^(let val r = ref "" in
                                       app (
                                        fn e => r := (!r)^(if !r = "" then ""
                                                                      else ",")^(stringof_value e)
                                       ) l; !r end)^")"
  | stringof_value' (VOP s) = s
  | stringof_value' _ = "<null>"

fun descrtype bool_t = "bool"
  | descrtype int_t = "int"
  | descrtype string_t = "string"
  | descrtype (parms_t(t::[])) = descrtype t
  | descrtype (parms_t l) =
     let val r = ref"" in
      app (fn t => r := (!r)^(if !r = "" then "" else ",")^(descrtype t)) l;
      "("^(!r)^")"
     end
  | descrtype (fun_t(t1,t2)) =
     (case t1 of
         fun_t _ => "("^(descrtype t1)^")"
       | _ => (descrtype t1))^" -> "^(descrtype t2)
  | descrtype (typevar(e,p)) =
     (case findtypeenv(typevar(e,p)) of
         typevar(e,p) =>
            (case !(nth(!e,p)) of
                noninit_t =>
                 let val pos = ref 0 val pr = ref 0 in
                  app (
                   fn t =>
                    (if !pr < p then
                      case !t of
                         noninit_t =>
                          (case findtypeenv(typevar(e,p)) of
                              typevar(e',p') =>
                               if nth(!e',p') = nth(!e,p) then inc pos else()
                            | _ =>
                              ())
                        | _ => ()
                     else();
                     inc pr)
                  ) (!e);
                  "'"^(chr(ord"a"+(!pos)))
                 end
              | t' =>
                 descrtype t')
       | t' =>
          descrtype t')
  | descrtype _ = "<bogus>"

and get_vartype (e,p) =
     findtypeenv (typevar(e,p))
              handle _ => typevar(e,p)

and set_vartype (e,p) typ =
     if is_initialized_type typ then
      settypeenv (e,p) typ
     else if not(is_initialized_type(!(nth(!e,p)))) then
      case typ of
         typevar(e',p') =>
          (* We need to equalize the var types and all references to the same *)
          if nth(!e,p) <> nth(!e',p') orelse (e,p) <> (e',p') then
           settypeenv (e,p) typ
          else
           ()
       | _ =>
          ()
     else
      ()

and subtracttype (parms_t l) typ = app (fn t => subtracttype t typ) l
```

```
  | subtracttype (fun_t(t1,t2)) typ = (subtracttype t1 typ; subtracttype t2 typ)
  | subtracttype (tv as typevar(e,p)) typ = subtypeenv (e,p) typ
  | subtracttype t' _ = ()
and subtypeenv tv typ =
     let fun subtypeinenv [] = []
           | subtypeinenv (Forall(sce,vt)::l) =
              if vt = tv then Forall(sube sce typ,vt)::l else Forall(sce,vt)::subtypeinenv l
     in
      set_current_env(subtypeinenv(get_current_env()))
     end
and findtypeenv typ =
     let fun findtypeinenv [] =
              raise TypingBug("type not found in env")
           | findtypeinenv (Forall(rl,vt)::l) =
              if typevar vt = typ orelse
               (* if two type vars specifiy the same reference,
                  they are the same *)
               (case vt of (e',p') =>
                    case typ of typevar(e,p) => nth(!e',p') = nth(!e,p))
              then
               case rl of
                  a::_ => a
                | _ => typevar vt
              else findtypeinenv l
     in
      case typ of
         parms_t l => parms_t(map findtypeenv l)
       | fun_t(t1,t2) => fun_t(findtypeenv t1,findtypeenv t2)
       | typevar _ =>
          (findtypeinenv(get_current_env()) handle _ => typ)
       | t' => t'
     end
and settypeenv tv typ =
     let fun settypeinenv [] =
              raise TypingBug"internal error: type not found in env!"
           | settypeinenv (Forall(rl,vt)::l) =
              if vt = tv then
               case rl of
                  [] => Forall(typ::[],vt)::l
                | a::_ =>
                   if vtcomp(a,typ) then Forall(typ::rl,vt)::l else
                    raise TypingBug("between "^(descrtype a)^" and "^(descrtype typ))
              else
               Forall(rl,vt)::settypeinenv l
     in
      set_current_env(settypeinenv(get_current_env()))
     end

and dispatch (venv,s) =
     let val ids = map (fn(id,_) => id) (!venv) in
      if s in_l ids then
       let val ret = ref VNULL
           val fnd = ref false
       in
        app (
         fn (i,(v,_)) =>
          if not(!fnd) then
           if i=s then (fnd := true; ret := !v) else()
          else()
        ) (rev(!venv));
        !ret
       end
      else raise(SUnbound s)
     end

and vtypeofv (VB _) = bool_t
  | vtypeofv (VI _) = int_t
  | vtypeofv (VS _) = string_t
  | vtypeofv (VPARMS (v::[])) = vtypeofv v
  | vtypeofv (VPARMS l) = parms_t(map vtypeofv l)
  | vtypeofv (VVAR(e,p)) = typeofvar(e,p)
  | vtypeofv (VAPP(v1,v2)) =
     let val t1 = vtypeofv v1 and t2 = vtypeofv v2 in
      case t1 of
         fun_t(t11,t12) => if vtcomp(t11,t2) then t12 else
                           raise TypingBug("the types "^(descrtype t11)^" -> "^
                                  (descrtype t12)^" and "^(descrtype t2)^" are not compatible")
```

```
        | typevar(e,p) =>
           (case !(nth(!e,p)) of
               noninit_t => fun_t(t1,t2)
             | t' => fun_t(t',t2))
        | noninit_t => fun_t(t1,t2)
        | _ => raise TypingBug("the types "^(descrtype t1)^" and "^(descrtype t2)^" are not compatible")
       end
   | vtypeofv (VBRANCH(_,v1,_)) =
      vtypeofv v1
   | vtypeofv (VFUN(_,t)) = t
   | vtypeofv _ = noninit_t
 and typeofvar (e,p) =
      (case nth(!e,p) of
          (_,(_,t)) =>
           case !t of
              typevar(e,p) =>
               (case !(nth(!e,p)) of
                   noninit_t => typevar(e,p)
                 | t' => t')
            | t' => t')
 and vtcomp(noninit_t,_) = true
   | vtcomp(_,noninit_t) = true
   | vtcomp(typevar(e1,p1),t2) =
      vtcomp(case findtypeenv(typevar(e1,p1)) of
                tv1' as typevar(e1',p1') =>
                 if nth(!e1,p1) = nth(!e1',p1') then noninit_t else tv1'
              | t1' => t1',
             t2)
   | vtcomp(t1,typevar(e2,p2)) =
      vtcomp(t1,
             case findtypeenv(typevar(e2,p2)) of
                 tv2' as typevar(e2',p2') =>
                  if nth(!e2,p2) = nth(!e2',p2') then noninit_t else tv2'
               | t2' => t2')
   | vtcomp (parms_t l1,parms_t l2) =
      if (length l1) = (length l2) then
       let val ret = ref true and lr = ref l1 and rr = ref l2 in
        while !ret andalso (case !lr of [] => false | _ => true) do
         if ((vtcomp(hd(!lr),hd(!rr))) handle _ => false)
          then (lr := tl(!lr); rr := tl(!rr)) else ret := false;
        !ret
       end
      else
       false
   | vtcomp (fun_t(fun_t(t1,t2),t3),t4) =
        ((vtcomp(t1,t3)) handle _ => false) andalso
            ((vtcomp(t2,t4)) handle _ => false)
   | vtcomp (t1,fun_t(fun_t(t2,t3),t4)) =
        ((vtcomp(t2,t4)) handle _ => false) andalso
            ((vtcomp(t1,t3)) handle _ => false)
   | vtcomp (fun_t(t1,t2),fun_t(t3,t4)) =
        ((vtcomp(t1,t3)) handle _ => false) andalso
            ((vtcomp(t2,t4)) handle _ => false)
   | vtcomp (t1 as fun_t _,t2) =
      (case t1 of
          fun_t(t11,t12) =>
           (case t2 of
               fun_t(t21,t22) => vtcomp(t11,t21) andalso vtcomp(t12,t22)
             | _ => vtcomp(t11,t2))
        | _ => t1 = t2)
   | vtcomp (t1,t2 as fun_t _) =
      (case t2 of
          fun_t(t21,t22) =>
           (case t1 of
               fun_t(t11,t12) => vtcomp(t21,t11) andalso vtcomp(t22,t12)
             | _ => vtcomp(t21,t1))
        | _ => t1 = t2)
   | vtcomp(t1,t2) = t1 = t2

  and occursintype (e,p) (parms_t(a::[])) =
       occursintype (e,p) a
    | occursintype (e,p) (parms_t(a::l)) =
       occursintype (e,p) a orelse occursintype (e,p) (parms_t l)
    | occursintype (e,p) (fun_t(t1,t2)) =
       occursintype (e,p) t1 orelse occursintype (e,p) t2
    | occursintype (e1,p1) (typevar(e2,p2)) =
       nth(!e1,p1) = nth(!e2,p2) orelse
```

```
       (case findtypeenv(!(nth(!e2,p2))) of
           typevar(e2',p2') =>
            if nth(!e2',p2') <> nth(!e2,p2)
             then occursintype (e1,p1) (typevar(e2',p2')) else false
         | t' => occursintype (e1,p1) t')
   | occursintype t1 t2 =
      false

(*
 * The operators for the local var env,
 *  that will be defined at the end
 *)

val make_venv = fn env => map (fn (a,b) => (a,(ref b,ref(vtypeofv b)))) env

val make_vvar = fn (env,id) =>
     (env := (!env)@make_venv[(id,VNULL)];
      case nth(!env,length(!env)-1) of
         (_,(_,rt)) => rt := make_typevar();
      VVAR(env,length(!env)-1))

(*
 * The value normalization functions
 *)

fun vnormalize venv (VPARMS(v::[])) = vnormalize venv v
  | vnormalize venv (VPARMS l) =
     let val r = ref[] in app (fn v => r := (!r)@[(vnormalize venv v)]) l; VPARMS(!r) end
  | vnormalize venv (VAPP(v1,v2)) = VAPP(vnormalize venv v1,vnormalize venv v2)
  | vnormalize venv (VFNDEF matchlist) =
     VFNDEF(map (fn (a,b) => (vnormalizel venv a,vnormalize venv b)) matchlist)
  | vnormalize venv (VID i) =
     let val id = stringof_ident i
         val p = ref(length(!venv)-1)
         val pos = ref ~1
         val f = ref VNULL
     in
      app (
       fn (a,(b,_)) =>
        if !pos = ~1 andalso a = id then
         (pos := !p;
          f := !b;
          case !f of
            VNULL (* left-hand-side variable *) => f := VVAR(venv,!pos)
          | _ (* !b was a proper value *) => ())
        else dec p
      ) (rev(!venv));
      case !f of
         VNULL => raise SUnbound id
       | _ => !f
     end
  | vnormalize venv (VALLIST l) =
     vnormalizelist venv (map (vnormalize venv) l)
  | vnormalize _ v = v
and vnormalizel venv (VPARMS(v::[])) = vnormalizel venv v
  | vnormalizel venv (VPARMS l) =
     let val r = ref[] in app (fn v => r := (!r)@[(vnormalizel venv v)]) l; VPARMS(!r) end
  | vnormalizel venv (VAPP(v1,v2)) =
     raise TypingBug"application on left-hand-side"
  | vnormalizel venv (VFNDEF matchlist) =
     raise TypingBug"function definition on left-hand-side"
  | vnormalizel venv (VID i) =
     make_vvar(venv,stringof_ident i)
  | vnormalizel venv (VALLIST l) =
     raise TypingBug"infix expression on left-hand-side"
  | vnormalizel venv v = v

and vnormalizelist venv (v::[]) = vnormalize venv v
  | vnormalizelist venv (v1::op1::v2::[]) =
     (case vnormalize venv op1 of
         VOP s1 =>
          if is_rssml_op s1 then
           let val o1' = dispatch(venv,s1)
               val v1' = vnormalize venv v1
               val v2' = vnormalize venv v2
           in
            vnormalize venv (VAPP(o1',VPARMS[v1',v2']))
```

```
           end
          else
           vnormalize venv (VPARMS(v1::op1::v2::[]))
       | op1' =>
          vnormalize venv (VPARMS(v1::op1'::v2::[])))
  | vnormalizelist venv (v1::op1::v2::op2::v3::l) =
     (case vnormalize venv op1 of
         VOP s1 =>
          (case vnormalize venv op2 of
              VOP s2 =>
               if is_rssml_op s1 andalso is_rssml_op s2 then
                if prio_rssml_op s1 >= prio_rssml_op s2 then
                 let val o1' = dispatch(venv,s1)
                     val v1' = vnormalize venv v1
                     val v2' = vnormalize venv v2
                     val op1ap = vnormalize venv (VAPP(o1',VPARMS[v1',v2']))
                 in
                  vnormalizelist venv (op1ap::op2::v3::l)
                 end
                else
                 vnormalizelist venv (v1::op1::vnormalizelist venv (v2::op2::v3::l)::[])
               else
                vnormalize venv (VPARMS(v1::op1::v2::op2::v3::l))
            | _ =>
              vnormalize venv (VPARMS((v1::op1::v2::op2::v3::l))))
       | _ =>
          vnormalize venv (VPARMS((v1::op1::v2::op2::v3::l))))

fun veq (v1,v2) =
     let val (v1',v2') = (vnormalize local_venv v1,vnormalize local_venv v2)
         val (t1,t2) = (vtypeofv v1,vtypeofv v2) in
      if vtcomp(t1,t2) then veq'(v1,v2) else
       raise TypingBug((descrtype t1)^" compared with "^(descrtype t2))
     end
and veq' (VB b1,VB b2) = b1 = b2
  | veq' (VI i1,VI i2) = i1 = i2
  | veq' (VS s1,VS s2) = s1 = s2
  | veq' (VPARMS l1,VPARMS l2) =
     length l1 = length l2 andalso
      let val ret = ref true
          val l1r = ref l1
      in
       app (
        fn l2e =>
         (if !ret then
           if not(veq(hd(!l1r),l2e)) then ret := false else()
          else();
          l1r := tl(!l1r))
       ) l2;
       !ret
      end
  | veq' _ = raise SUnbound"operator \"=\" not implemented for these types"
and vlt (v1,v2) =
     let val (v1',v2') = (vnormalize local_venv v1,vnormalize local_venv v2)
         val (t1,t2) = (vtypeofv v1',vtypeofv v2')
     in
      if vtcomp(t1,t2) then vlt'(v1',v2') else
       raise TypingBug((descrtype t1)^" compared with "^(descrtype t2))
     end
and vlt' (VB b1,VB b2) = not b1 andalso b2
  | vlt' (VI i1,VI i2) = i1 < i2
  | vlt' (VS s1,VS s2) = s1 < s2
  | vlt' (VPARMS l1,VPARMS l2) =
     length l1 = length l2 andalso
      let val ret = ref true
          val l1r = ref l1
      in
       app (
        fn l2e =>
         (if !ret then
           if not(vlt(hd(!l1r),l2e)) then ret := false else()
          else();
          l1r := tl(!l1r))
       ) l2;
       !ret
      end
  | vlt' _ = raise SUnbound"operator \"<\" not implemented for these types"
```

```
and vle (v1,v2) =
     let val (v1',v2') = (vnormalize local_venv v1,vnormalize local_venv v2)
         val (t1,t2) = (vtypeofv v1',vtypeofv v2')
     in
      if vtcomp(t1,t2) then vle'(v1',v2') else
       raise TypingBug((descrtype t1)^" compared with "^(descrtype t2))
     end
and vle' (VB b1,VB b2) = not b1 orelse b2
  | vle' (VI i1,VI i2) = i1 <= i2
  | vle' (VS s1,VS s2) = s1 <= s2
  | vle' (VPARMS l1,VPARMS l2) =
     length l1 = length l2 andalso
      let val ret = ref true
          val l1r = ref l1
      in
       app (
        fn l2e =>
         (if !ret then
           if not(vle(hd(!l1r),l2e)) then ret := false else()
          else();
          l1r := tl(!l1r))
       ) l2;
       !ret
      end
  | vle' _ = raise SUnbound"operator \"<=\" not implemented for these types"
and vgt (v1,v2) =
     let val (v1',v2') = (vnormalize local_venv v1,vnormalize local_venv v2)
         val (t1,t2) = (vtypeofv v1',vtypeofv v2')
     in
      if vtcomp(t1,t2) then vgt'(v1',v2') else
       raise TypingBug((descrtype t1)^" compared with "^(descrtype t2))
     end
and vgt' (VB b1,VB b2) = b1 andalso not b2
  | vgt' (VI i1,VI i2) = i1 > i2
  | vgt' (VS s1,VS s2) = s1 > s2
  | vgt' (VPARMS l1,VPARMS l2) =
     length l1 = length l2 andalso
      let val ret = ref true
          val l1r = ref l1
      in
       app (
        fn l2e =>
         (if !ret then
           if not(vgt(hd(!l1r),l2e)) then ret := false else()
          else();
          l1r := tl(!l1r))
       ) l2;
       !ret
      end
  | vgt' _ = raise SUnbound"operator \">\" not implemented for these types"
and vge (v1,v2) =
     let val (v1',v2') = (vnormalize local_venv v1,vnormalize local_venv v2)
         val (t1,t2) = (vtypeofv v1',vtypeofv v2')
     in
      if vtcomp(t1,t2) then vge'(v1',v2') else
       raise TypingBug((descrtype t1)^" compared with "^(descrtype t2))
     end
and vge' (VB b1,VB b2) = b1 orelse not b2
  | vge' (VI i1,VI i2) = i1 >= i2
  | vge' (VS s1,VS s2) = s1 >= s2
  | vge' (VPARMS l1,VPARMS l2) =
     length l1 = length l2 andalso
      let val ret = ref true
          val l1r = ref l1
      in
       app (
        fn l2e =>
         (if !ret then
           if not(vge(hd(!l1r),l2e)) then ret := false else()
          else();
          l1r := tl(!l1r))
       ) l2;
       !ret
      end
  | vge' _ = raise SUnbound"operator \">=\" not implemented for these types"
and vne (v1,v2) = not(veq(v1,v2))
```

```
 fun evopa (oper,v1,v2) =
      let val v1' = case vnormalize local_venv v1 of
                      VI _ => v1
                    | VVAR(e,p) => case nth(!e,p) of (_,(v,_)) => !v
          val v2' = case vnormalize local_venv v2 of
                      VI _ => v2
                    | VVAR(e,p) => case nth(!e,p) of (_,(v,_)) => !v
      in
       case v1' of VI i1 =>
           (case v2' of VI i2 =>
                VI(case oper of "+" => i1+i2 | "-" => i1-i2 | "*" => i1*i2 | "/" => i1 div i2)
             | _ => raise TypingBug(oper^" applied to "^(descrtype(vtypeofv v1'))^"*"^
                                                        (descrtype(vtypeofv v2'))))
        | _ => raise TypingBug(oper^" applied to "^(descrtype(vtypeofv v1'))^"*"^
                                                   (descrtype(vtypeofv v2')))
      end

 (* the infix operators ... *)
 val init_venv =
  [("+",VFUN(fn VPARMS[v1,v2] => evopa("+",v1,v2),fun_t(parms_t[int_t,int_t],int_t))),
   ("-",VFUN(fn VPARMS[v1,v2] => evopa("-",v1,v2),fun_t(parms_t[int_t,int_t],int_t))),
   ("*",VFUN(fn VPARMS[v1,v2] => evopa("*",v1,v2),fun_t(parms_t[int_t,int_t],int_t))),
   ("/",VFUN(fn VPARMS[v1,v2] => evopa("/",v1,v2),fun_t(parms_t[int_t,int_t],int_t))),
   ("=",VFUN(fn VPARMS[v1,v2] => VB(veq(v1,v2)),
             fun_t(let val nt = (reset_vartypes(); make_typevar()) in parms_t[nt,nt] end,bool_t))),
   ("<",VFUN(fn VPARMS[v1,v2] => VB(vlt(v1,v2)),
             fun_t(let val nt = (reset_vartypes(); make_typevar()) in parms_t[nt,nt] end,bool_t))),
   ("<=",VFUN(fn VPARMS[v1,v2] => VB(vle(v1,v2)),
              fun_t(let val nt = (reset_vartypes(); make_typevar()) in parms_t[nt,nt] end,bool_t))),
   (">",VFUN(fn VPARMS[v1,v2] => VB(vgt(v1,v2)),
             fun_t(let val nt = (reset_vartypes(); make_typevar()) in parms_t[nt,nt] end,bool_t))),
   (">=",VFUN(fn VPARMS[v1,v2] => VB(vge(v1,v2)),
              fun_t(let val nt = (reset_vartypes(); make_typevar()) in parms_t[nt,nt] end,bool_t))),
   ("<>",VFUN(fn VPARMS[v1,v2] => VB(vne(v1,v2)),
              fun_t(let val nt = (reset_vartypes(); make_typevar()) in parms_t[nt,nt] end,bool_t)))]


 (* Initializing the local var env *)

 val _ = local_venv := make_venv init_venv

end (* RSsml *)
```

A.2 Mini ML Parser

A.2.1 Lexical Analyser

The lexical analyser of this Section must be translated into pure SML code using `sml-lex`.

```
(*
 * RSsml.lex
 *
 * Raddstar Structured Meta Language
 * Expression Compiler -- Scanner
 *
 * Copyright Martin Steeg, 1996 - 1999
 *)

structure Tokens = Tokens

type pos = int
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult = (svalue,pos) token

val pos = ref 0
val eof = fn () => Tokens.EOF(!pos,!pos)

structure KeyWord : sig
                        val find : string ->
                                (int * int -> (svalue,int) token) option
                    end =
  struct

        val TableSize = 211
        val HashFactor = 5

        val hash = fn s =>
           fold (fn (c,v)=>(v*HashFactor+(ord c)) mod TableSize) (explode s) 0


        val HashTable = Array.array(TableSize,nil) :
                  (string * (int * int -> (svalue,int) token)) list Array.array


        val add = fn (s,v) =>
         let val i = hash s
         in Array.update(HashTable,i,(s,v) :: (Array.sub(HashTable, i)))
         end

        val find = fn s =>

          let val i = hash s
              fun f ((key,v)::r) = if s=key then SOME v else f r
                | f nil = NONE
          in  f (Array.sub(HashTable, i))
          end

        open Tokens

        val _ =
            (List.app add
                [
                 ("fun",FUN),
                 ("val",VAL),
                 ("fn",FN),
                 ("if",IF),
                 ("then",THEN),
                 ("else",ELSE),
                 ("fi",FI)
                ])
   end

%%
%header (functor RSsml_ParserLexFun(structure Tokens : RSsml_Parser_TOKENS));

%s C COMMENT;
alpha=[A-Za-z_\.#$];
```

```
digit=[0-9];
ws = [\ \t];

%%

<INITIAL>\^C                          => (raise LrParser.ParseError);
<INITIAL>\^D                          => (Tokens.EOF(!pos,!pos));
<INITIAL>\n                           => (inc pos; RSsml.psln2(); lex());
<INITIAL>\"[^"]*\"                    => (let val tok = substring(yytext,1,(String.length yytext)-2)
                                          in Tokens.STRING(tok,!pos,!pos) end);
<INITIAL>{ws}+                        => (lex());
<INITIAL>"("                          => (Tokens.LPAR(!pos,!pos));
<INITIAL>","                          => (Tokens.COMMA(!pos,!pos));
<INITIAL>")"                          => (Tokens.RPAR(!pos,!pos));
<INITIAL>"+"                          => (Tokens.OP("+",!pos,!pos));
<INITIAL>"-"                          => (Tokens.OP("-",!pos,!pos));
<INITIAL>"*"                          => (Tokens.OP("*",!pos,!pos));
<INITIAL>"(*"                         => (YYBEGIN C; lex());
<INITIAL>"/"                          => (Tokens.OP("/",!pos,!pos));
<INITIAL>"<"                          => (Tokens.OP("<",!pos,!pos));
<INITIAL>"<="                         => (Tokens.OP("<=",!pos,!pos));
<INITIAL>">"                          => (Tokens.OP(">",!pos,!pos));
<INITIAL>">="                         => (Tokens.OP(">=",!pos,!pos));
<INITIAL>"<>"                         => (Tokens.OP("<>",!pos,!pos));
<INITIAL>":="                         => (Tokens.OP(":=",!pos,!pos));
<INITIAL>"!"                          => (Tokens.IDENT(RSsml.make_ident"!",!pos,!pos));
<INITIAL>\;                           => (Tokens.SEMI(!pos,!pos));
<INITIAL>"=>"                         => (Tokens.IMPLIES(!pos,!pos));
<INITIAL>"="                          => (Tokens.EQ(!pos,!pos));
<INITIAL>"|"                          => (Tokens.BAR(!pos,!pos));
<INITIAL>"_"                          => (Tokens.UNDERBAR(!pos,!pos));

<INITIAL>{digit}+                     => (Tokens.INT
                                          (revfold (fn (a,r) => ord(a)-ord("0")+10*r)
                                                   (explode yytext) 0,
                                            !pos,!pos));

<INITIAL>{alpha}({alpha}|{digit})*    => (if RSsml.is_rssml_op yytext then
                                           Tokens.OP(yytext,!pos,!pos)
                                          else
                                           case KeyWord.find yytext of
                                              SOME kwv => kwv(!pos,!pos)
                                            | _ =>
                                              let val new = ref false
                                                  val tnm = RSsml.make_ident yytext
                                                   handle (RSsml.SUnbound _) =>
                                                     (new := true; RSsml.make_ident yytext)
                                              in
                                               Tokens.IDENT(tnm,!pos,!pos)
                                              end);

<INITIAL>.                            => (RSsml.say("RSsml: ignoring bad character \"^yytext^"\"");
                                          lex());
<C>\n+                                => (pos := (!pos) + (String.length yytext); lex());
<C>[^()*\n]+                          => (lex());
<C>"(*"                               => (lex());
<C>"*)"                               => (YYBEGIN INITIAL; lex());
<C>[*()]                              => (lex());
```

A.2.2 Parser

The parser of this Section must be translated into pure SML code using `sml-yacc`.

```
(* RSsml.grm, Raddstar Structured Meta Language Expression Compiler -- Parser *)

%%

%eop EOF SEMI

%pos int

%left NOT
%left OP
%left EQ

%term     EOF | SEMI
        | NOT | OP of string | EQ
        | UNDERBAR | BOOL of bool | INT of int | STRING of string
        | IDENT of RSsml.ident
        | LPAR | COMMA | RPAR
        | IF | THEN | ELSE | FI
        | FUN | VAL | FN | IMPLIES | BAR

%nonterm  START of RSsml.stmt
        | FUNDEF of RSsml.stmt
         | FUNMATCHLIST of (RSsml.value * RSsml.value) list
         | FUNMATCH of RSsml.value * RSsml.value
        | VALDEF of RSsml.stmt
        | APPL of RSsml.value | APII of RSsml.value | APARMS of RSsml.value list
        | VALUE of RSsml.value | PARMS of RSsml.value list
        | OPER of string
        | FNDEF of RSsml.value
         | FNMATCHLIST of (RSsml.value * RSsml.value) list
         | FNMATCH of RSsml.value * RSsml.value

%name RSsml_Parser
%header (functor RSsml_ParserLrVals(structure Token : TOKEN))

%noshift EOF

%%

START   : FUNDEF                        (FUNDEF)
        | VALDEF                        (VALDEF)
        | APPL                          (RSsml.VALDEF(RSsml.IT,APPL))
        |                               (RSsml.NULLSTMT)

FUNDEF  : FUN IDENT FUNMATCHLIST     (RSsml.FUNDEF(IDENT,FUNMATCHLIST))

FUNMATCHLIST:
          FUNMATCH                      ([FUNMATCH])
        | FUNMATCHLIST BAR FUNMATCH     (FUNMATCHLIST@[FUNMATCH])

FUNMATCH: VALUE EQ APPL                 (VALUE,APPL)

VALDEF  : VAL IDENT EQ APPL             (RSsml.VALDEF(IDENT,APPL))

APPL    : APII                          (APII)
        | FNDEF                         (FNDEF)
        | IF APPL THEN APPL ELSE APPL FI(RSsml.VBRANCH(APPL1,APPL2,APPL3))
        | APPL APII                     (RSsml.VAPP(APPL,APII))
        | APPL OPER APPL                (case APPL1 of
                                             RSsml.VALLIST l1 =>
                                                 RSsml.VALLIST(l1@[RSsml.VOP OPER]@(case APPL2 of
                                                                                      RSsml.VALLIST l2 => l2
                                                                                    | _ => [APPL2]))
                                           | ap => RSsml.VALLIST([ap]@[RSsml.VOP OPER]@(case APPL2 of
                                                                                          RSsml.VALLIST l2 => l2
                                                                                        | _ => [APPL2])))

APARMS  : APPL                          ([APPL])
        | APARMS COMMA APPL             (APARMS@[APPL])

APII    : VALUE                         (VALUE)
        | LPAR APARMS RPAR              (RSsml.VPARMS APARMS)
```

```
VALUE   : IDENT                         (RSsml.VID IDENT)
        | BOOL                          (RSsml.VB BOOL)
        | INT                           (RSsml.VI INT)
        | STRING                        (RSsml.VS STRING)
        | UNDERBAR                      (RSsml.VNULL)
        | LPAR PARMS RPAR               (RSsml.VPARMS PARMS)

OPER    : OP                            (OP)
        | EQ                            ("=")

PARMS   : VALUE                         ([VALUE])
        | PARMS COMMA VALUE             (PARMS@[VALUE])

FNDEF   : FN FNMATCHLIST                (RSsml.VFNDEF FNMATCHLIST)

FNMATCHLIST:
          FNMATCH                       ([FNMATCH])
        | FNMATCHLIST BAR FNMATCH       (FNMATCHLIST@[FNMATCH])

FNMATCH : VALUE IMPLIES APPL            (VALUE,APPL)
```

A.3 The Mini ML Compiler

A.3.1 The Compiler/Expression Evaluator

```
(*
 * RSsmlComp.sml
 *
 * Raddstar Structured Meta Language
 * Expression Compiler
 *
 * Copyright Martin Steeg, 1996 - 1999
 *)

signature RSSMLCOMP =
 sig
  type stmt
  val compile : stmt -> bool
 end

structure RSsmlComp : RSSMLCOMP =
 struct

  open RSsml

  infix 9 in_l

  fun set_type_type (parms_t l) typ2 =
       (case typ2 of
           parms_t l2 =>
            if length l = length l2 then
             let val tr = ref l2 in
              app (fn t => (set_type_type t (hd(!tr)); tr := tl(!tr))) l
             end
            else
             raise TypingBug("types "^(descrtype(parms_t l))^" and "^(descrtype(parms_t l2))^
                                    " are not compatible")
         | _ =>
            raise TypingBug("types "^(descrtype(parms_t l))^" and "^(descrtype typ2)^
                                    " are not compatible"))
   | set_type_type (fun_t(t1,t2)) typ2 =
      (case typ2 of
          fun_t(t21,t22) =>
           (set_type_type t1 t21; set_type_type t2 t22)
         | _ =>
            raise TypingBug("types "^(descrtype(fun_t(t1,t2)))^" and "^(descrtype typ2)^
                                    " are not compatible"))

   | set_type_type (tv as typevar(e1,p1)) typ2 =
       (case typ2 of
           typevar(e2,p2) =>
            if nth(!e2,p2) <> nth(!e1,p1) then set_vartype (e2,p2) tv else()
         | _ =>
            set_vartype (e1,p1) typ2)
   | set_type_type typ1 typ2 =
       if vtcomp(typ1,typ2) then()
       else
        raise TypingBug("types "^(descrtype typ1)^" and "^(descrtype typ2)^" are not compatible")



  (*
   * The implementation of the type inference algorithm
   *
   * The functions of this coding are borrowed from Mauny 1993 and Mauny 1995
   *)

  fun shorten t = shorten'(rtnormalize t)
  and shorten' (parms_t l) = parms_t(map shorten l)
    | shorten' (fun_t(t1,t2)) = fun_t(shorten t1,shorten t2)
    | shorten' (tv as typevar(e,p)) =
       (case shorten(!(nth(!e,p))) of
           typevar(e',p') =>
            (shorten (!(nth(!e',p'))));
             e :=
              map (
```

```
             fn rt => if rt = nth(!e,p) then nth(!e',p') else rt
            ) (!e);
           tv)
       | _ => tv)
  | shorten' t = t
and generalizetype (gamma,tau) =
     Forall(map typevar (subl (varsoftype(typevar tau)) (unknownsofenv gamma)),tau)
and geninstance (Forall(gv,tau)) =
     let fun ginstance (parms_t l) = parms_t(map ginstance l)
           | ginstance (fun_t(t1,t2)) = fun_t(ginstance t1,ginstance t2)
           | ginstance (tv as typevar(e,p)) =
              (case !(nth(!e,p)) of
                  parms_t l => parms_t(map ginstance l)
                | fun_t(t1,t2) => fun_t(ginstance t1,ginstance t2)
                | noninit_t => findtypeenv tv
                | t' => ginstance t')
           | ginstance noninit_t = raise TypingBug"geninstance"
           | ginstance t' = t'
     in
      ginstance(typevar tau)
     end
and typesynthi (env as (_,venv),gamma) =
     fn VPARMS l =>
         unify(parms_t(map (typesynthi (env,gamma)) l),
               parms_t(rev(map (typesynthi (env,gamma)) (rev l))))
      | VVAR(e,p) =>
         (case nth(!e,p) of (n,(_,rt)) =>
              (case !rt of
                  typevar(e',p') => findtypeenv(typevar(e',p'))
                | t => ltnormalize t))
      | VAPP(v1,v2) =>
         let val vt2 = typesynthi (env,gamma) v2 in
          case (ltnormalize(unify(typesynthi (env,gamma) v1,
                       fun_t(vt2,make_typevar()))
                 handle _ => raise TypingBug((descrtype(vtypeofv v1))^" applied to "^
                                             (descrtype(vtypeofv v2)))) of
             fun_t(t1,t2) =>
              (set_type_type t1 (ltnormalize vt2);
               t2)
           | t' =>
              raise TypingBug("suspicious evaluation of unify ["^(descrtype t')^"]")
         end
      | VBRANCH(p,v1,v2) =>
         let val pt = typesynthi (env,gamma) p
             val v1t = typesynthi (env,gamma) v1
             val v2t = typesynthi (env,gamma) v2
         in
          (unify(pt,bool_t); unify(v1t,v2t))
            handle _ =>
             raise TypingBug("if-then-else-fi: (bool,'a,'a) -> 'a used with ("^(descrtype pt)^
                             ","^(descrtype v1t)^","^(descrtype v2t)^")")
         end
      | VFNDEF fl =>
         let val fl' = map (fn (l,r) => (levaluate env l,revaluate env r)) fl in
          typesynthi' (env,gamma) fl'
         end
      | v' => vtypeofv v'
and typesynthi' (env,gamma) fl =
      let val (lt,rt) = (ref noninit_t,ref noninit_t) in
       app (
        fn (l,r) =>
         let val vt = case make_typevar() of typevar(e,p) => (e,p)
             val (ltyp,rtyp) = (typesynthi (env,gamma@[(Forall([],vt))]) l,
                                typesynthi (env,gamma@[(Forall([],vt))]) r)
             val ltyp' = unify(!lt,ltyp)
                  handle _ => raise TypingBug("between "^(descrtype(fun_t(!lt,!rt)))^" and "^
                                              (descrtype(fun_t(ltyp,rtyp))))
             val rtyp' = unify(!rt,rtyp)
                  handle _ => raise TypingBug("between "^(descrtype(fun_t(!lt,!rt)))^" and "^
                                              (descrtype(fun_t(ltyp,rtyp))))
         in
          lt := ltyp'; rt := rtyp'
         end
       ) fl;
       fun_t(!lt,!rt)
      end
```

```
and unify (tau1,tau2) =
     (case (shorten tau1,shorten tau2) of
         (noninit_t,t2) => t2
       | (t1,noninit_t) => t1
       | (bool_t,bool_t) => bool_t
       | (int_t,int_t) => int_t
       | (string_t,string_t) => string_t
       | (parms_t l1,parms_t l2) =>
          if length l1 <> length l2 then
           raise TypingBug("between "^(descrtype(parms_t l1))^" and "^(descrtype(parms_t l2)))
          else
           (let val l1r = ref l1 val ret = ref[] in
             app (
              fn l2e =>
               (ret := (!ret)@[(unify(hd(!l1r),l2e))]; l1r := tl(!l1r))
             ) l2;
             parms_t(!ret)
            end
             handle _ =>
              raise TypingBug("between "^(descrtype(parms_t l1))^" and "^(descrtype(parms_t l2))))
       | (fun_t(t11,t12),fun_t(t21,t22)) =>
          (fun_t(unify(t11,t21),unify(t12,t22))
            handle _ =>
             raise TypingBug("between "^(descrtype(fun_t(t11,t12)))^" and "^(descrtype(fun_t(t21,t22)))))
       | (tv1 as typevar(e1,p1),tv2 as typevar(e2,p2)) =>
          if nth(!e1,p1) <> nth(!e2,p2) then
           (e1 :=
             map (
              fn rt => if rt = nth(!e1,p1) then nth(!e2,p2) else rt
             ) (!e1);
            tv1)
          else tv1
       | (t1,tv2 as typevar(e2,p2)) =>
          if not(occursintype (e2,p2) t1) then
           (set_vartype (e2,p2) t1; tv2)
          else raise TypingBug("between "^(descrtype t1)^" and "^(descrtype tv2))
       | (tv1 as typevar(e1,p1),t2) =>
          if not(occursintype (e1,p1) t2) then
           (set_vartype (e1,p1) t2; tv1)
          else raise TypingBug("between "^(descrtype tv1)^" and "^(descrtype t2))
       | (t1,t2) =>
          raise TypingBug("between "^(descrtype t1)^" and "^(descrtype t2)))

(*
 * The RSsml specific type unification functions
 *)

and ltnormalize (parms_t(t::[])) = ltnormalize t
  | ltnormalize (fun_t(t1,t2)) =
     let val t1' = ltnormalize t1 and t2' = ltnormalize t2 in
      case t1' of
         fun_t(t11',_) =>
          if vtcomp(t11',t2') then
           let val nt2' = unify(t11',t2') handle _ => t2' in
            fun_t(t1',nt2')
           end
          else
           fun_t(t1',t2')
       | _ => fun_t(t1',t2')
     end
  | ltnormalize (typevar(e,p)) =
     (case nth(!e,p) of rt =>
          (case !rt of
              noninit_t => findtypeenv(typevar(e,p))
            | t => ltnormalize t))
  | ltnormalize t = t
and rtnormalize (parms_t(t::[])) = rtnormalize t
  | rtnormalize (fun_t(fun_t(t1,t2),t3)) =
     let val (ft',t3') = (rtnormalize(fun_t(t1,t2)),rtnormalize t3) in
      case ft' of
         fun_t(t1',t2') =>
          if vtcomp(t1',t3') then t2'
          else
           raise TypingBug("types "^(descrtype(fun_t(t1',t2')))^" and "^
                                    (descrtype t3')^" could not be unified")
       | _ =>
          fun_t(ft',t3')
```

```
     end
  | rtnormalize (fun_t(ityp,otyp)) = fun_t(rtnormalize ityp,rtnormalize otyp)
  | rtnormalize (typevar(e,p)) =
     (case nth(!e,p) of rt =>
          (case !rt of
              noninit_t => findtypeenv(typevar(e,p))
            | t => rtnormalize t))
  | rtnormalize t = t

and setvvar (e,p) v =
     (case nth(!e,p) of (_,(rv,_)) => rv := v)

(*
 * The value evaluation and normalization functions
 *)

and vevaluate (env as (_,venv)) v =
      vevaluate' env (revaluate env v)
and vevaluate' env (VPARMS l) = VPARMS(map (vevaluate' env) l)
  | vevaluate' _ (VVAR(e,p)) = (case nth(!e,p) of (_,(v,_)) => !v)
  | vevaluate' env (VAPP(v1,v2)) =
     let val v2' = vevaluate' env v2 in
      case vevaluate' env v1 of
         VFUN(f,t) =>
           f v2'
       | v1' =>
           raise TypingBug("value "^(stringof_value v1')^" applied to "^(stringof_value v2'))
     end
  | vevaluate' env (VBRANCH(p,v1,v2)) =
     (case vevaluate' env p of
         VB b => if b then vevaluate' env v1 else vevaluate' env v2)
  | vevaluate' env (VFNDEF l) = fnevaluate env l
  | vevaluate' env v = v
and levaluate (env as (_,venv)) v = levaluate' env (vnormalizel venv v)
and levaluate' _ (VB b) = VB b
  | levaluate' _ (VI i) = VI i
  | levaluate' _ (VS s) = VS s
  | levaluate' env (VPARMS l) = VPARMS(map (levaluate' env) l)
  | levaluate' _ (vv as VVAR _) = vv
  | levaluate' _ VNULL = VNULL
  | levaluate' _ v =
     raise TypingBug("unexpected left-hand-side value: "^(stringof_value v))
and revaluate (env as (_,venv)) v =
     revaluate' env (vnormalize venv v)
and revaluate' _ (VB b) = VB b
  | revaluate' _ (VI i) = VI i
  | revaluate' _ (VS s) = VS s
  | revaluate' env (VPARMS(v::[])) = revaluate env v
  | revaluate' env (VPARMS l) = VPARMS(map (revaluate env) l)
  | revaluate' env (VVAR(e,p)) =
     (case nth(!e,p) of
         (_,(v,_)) => case !v of VNULL => VVAR(e,p) | v' => v')
  | revaluate' env (VAPP(v1,v2)) = VAPP(revaluate env v1,revaluate env v2)
  | revaluate' env (VBRANCH(v1,v2,v3)) =
     VBRANCH(revaluate env v1,revaluate env v2,revaluate env v3)
  | revaluate' _ (f as VFUN _) = f
  | revaluate' env (VFNDEF l) = fnevaluate env l
  | revaluate' _ (VOP oper) = VOP oper
  | revaluate' _ VNULL = raise SUnbound"right-hand-side value: null"
and fevaluate env (VFUN(f,_),arg) = f(vevaluate env arg)
  | fevaluate env (VVAR(e,p),arg) =
     (case nth(!e,p) of
         (_,(ref(VFUN(f,t)),_)) =>
          fevaluate env (VFUN(f,t),arg)
       | (_,(ref v,_)) =>
          raise TypingBug((stringof_value v)^" applied to "^(stringof_value arg)))
  | fevaluate env (v,arg) =
     raise TypingBug((stringof_value v)^" applied to "^(stringof_value arg))
and fnevaluate (id,venv) fl =
     fnevaluate' (id,venv)
      let val varlist =
               map (
                fn (l,r) =>
                 let val venv' = ref(!venv) in
                  (levaluate (id,venv') l,revaluate (id,venv') r,venv')
                 end
               ) fl
```

```
          val (lt,rt) = (ref noninit_t,ref noninit_t)
      in
       (map(
         fn (lval,rval,venv') =>
          case typesynthi' ((id,venv),make_typesc(!venv')) [(lval,rval)] of
             fun_t(ltyp,rtyp) =>
              let val ltyp' = unify(!lt,ltyp)
                       handle _ => raise TypingBug("between "^(descrtype(fun_t(!lt,!rt)))^" and "^
                                                   (descrtype(fun_t(ltyp,rtyp))))
                  val rtyp' = unify(!rt,rtyp)
                       handle _ => raise TypingBug("between "^(descrtype(fun_t(!lt,!rt)))^" and "^
                                                   (descrtype(fun_t(ltyp,rtyp))))
              in
               lt := ltyp';
               rt := rtyp';
               (lval,rval)
              end
           | t' =>
              raise TypingBug("typesynthi' (VFNDEF ...) evaluates "^(descrtype t'))
        ) varlist,
        fun_t(!lt,!rt))
      end
and fnevaluate' (id,_) ([],_) = raise TypingBug"empty function list"
  | fnevaluate' env ((v1,v2)::l,fun_t(t1,t2)) =
     VFUN(fn v =>
           if unify2(v,v1)
            then vevaluate env v2 else fnevaluatep env (v,t1,l),fun_t(t1,t2))
and fnevaluatep (id,_) (_,_,[]) = raise UNcaught id
  | fnevaluatep env (v,t1,(v1,v2)::l) =
     if unify2(v,v1)
      then vevaluate env v2 else fnevaluatep env (v,t1,l)
and unify2 (v,v1) =
     unify2'(v,v1)
      handle _ => raise TypingBug((descrtype(vtypeofv v1))^" and "^
                                  (descrtype(vtypeofv v))^" are not compatible")
and unify2' (v,v1) =
     (case v1 of
         VB b1 =>
          (case v of
              VB b => b = b1
            | VVAR(e,p) => (setvvar (e,p) v1; true))
       | VI i1 =>
          (case v of
              VI i => i = i1
            | VVAR(e,p) => (setvvar (e,p) v1; true))
       | VS s1 =>
          (case v of
              VS s => s = s1
            | VVAR(e,p) => (setvvar (e,p) v1; true))
       | VPARMS l1 =>
          (case v of
              VPARMS l =>
               let val ret = ref true val l1r = ref l1 in
                app (
                 fn v =>
                  (if !ret
                    then ret := unify2(v,hd(!l1r)) else();
                   l1r := tl(!l1r))
                ) l;
                !ret
               end)
       | VVAR(e,p) => (setvvar (e,p) v; true)
       | VNULL => true
       | _ => raise TypingBug"warning: not yet implemented")


(*
 * The functions for the final type and value evaluation
 *)

fun set_valtype_env (env as (id,venv)) (va,typ) =
     let val (va',typ') = (vnormalize venv va,ltnormalize typ) in
      if not(vtcomp(vtypeofv va',typ')) then
       raise TypingBug("between "^(descrtype(vtypeofv va'))^" and "^(descrtype typ'))
      else
       set_valtype_env' env (va',typ)
     end
```

```
and set_valtype_env' env (va,typ) =
     case va of
        VPARMS l =>
         let val tr = ref(case (ltnormalize typ) of parms_t l => l) in
          VPARMS(map (fn v => set_valtype_env' env (v,let val t = hd(!tr) in tr := tl(!tr); t end)) l)
         end
      | _ =>
         if (case va of VFUN _ => true | VVAR _ => true | _ => false) then
          let val sce = get_current_env()
              val new = ref[]
              fun evalnew (sce,parms_t l) = app (fn t => evalnew(sce,t)) l
                | evalnew (sce,fun_t(t1,t2)) = (evalnew(sce,t1); evalnew(sce,t2))
                | evalnew (sce,tv as typevar(e,p)) =
                   let fun findtl [] = raise TypingBug"typevar not found in env"
                         | findtl (Forall(l,vt)::r) =
                            if vt = (e,p) then l
                            else if occursintype (e,p) (typevar vt) then []
                            else findtl r
                   in
                    (case findtl sce of
                        a::_ => evalnew(sce,a)
                      | _ =>
                         if tv <> typ andalso occursintype (e,p) typ
                         andalso not(tv in_l (map ! (!new)))
                          then new := (!new)@[(ref tv)] else())
                     handle _ => ()
                     (* ignore TypingBug, since typevar may have been re-assigned previously *)
                   end
                | evalnew _ = ()
              fun subst (parms_t l) = parms_t(map subst l)
                | subst (fun_t(t1,t2)) = fun_t(subst t1,subst t2)
                | subst (tv as typevar(e,p)) =
                   (case posl(map ! (!new),tv) of
                       ~1 => tv
                     | n => typevar(new,n))
                | subst t' = t'
              fun replace typ =
                   let val ret = subst typ in
                    app (fn tr as ref(typevar _) => tr := noninit_t | _ => ()) (!new);
                    ret
                   end
          in
           app (
            fn (Forall(l,vt)) => evalnew(sce,typevar vt)
           ) sce;
           case va of
              VFUN(f,_) => VFUN(f,replace typ)
            | VVAR(e,p) =>
               case nth(!e,p) of (_,(_,rt)) => (rt := replace typ; va)
          end
         else (* nothing to do *)
          va

fun headt (fun_t(t1,_)) = t1
  | headt t = t
and tailt (fun_t(_,t2)) = t2
  | tailt t = t
and firstt (fun_t(t1,fun_t(t2,t3))) = fun_t(t1,firstt(fun_t(t2,t3)))
  | firstt (fun_t(t1,t2)) = t1
  | firstt t = t
and lastt (fun_t(_,t2)) = lastt t2
  | lastt t = t

(*
 * The Z-Fixpoint combinator implements recursive function evaluation
 *)

val fix =
     VFUN(fn f =>
           case VFUN(fn x =>
              case vevaluate (FIX,local_venv) x of VFUN(x,xt) =>
                 case vevaluate (FIX,local_venv) f of VFUN(f,ft) =>
                    f(VFUN(fn VFUN(z,zt) =>
                       case x(VFUN(x,xt)) of
                          VFUN(y,_) => y(VFUN(z,zt)),
                           headt ft)),
                vtypeofv f)
```

```
            of VFUN(y,yt) =>
             y(VFUN(fn x =>
                case vevaluate (FIX,local_venv) x of VFUN(x,xt) =>
                   case vevaluate (FIX,local_venv) f of VFUN(f,ft) =>
                      f(VFUN(fn VFUN(z,zt) =>
                         case x(VFUN(x,xt)) of
                            VFUN(y,_) => y(VFUN(z,zt)),
                             headt ft)),
                    headt yt)),
          fun_t(fun_t(noninit_t,noninit_t),fun_t(noninit_t,noninit_t)))

fun compile (FUNDEF(i,l)) =
     let val _ = reset_vartypes()
         val newfun = stringof_ident i
         val newvenv = ref(!local_venv)
         val newpos = length(!newvenv)
              (* newpos will be the position of newfun in the newvenv *)
         val f = make_vvar(newvenv,newfun)
         val l' = map (fn (l,r) => (levaluate (i,newvenv) l,revaluate (i,newvenv) r)) l
         val t' = typesynthi' ((i,newvenv),make_typesc(!newvenv)) l'
     in
      case vevaluate (i,newvenv) (VAPP(fix,VFNDEF[(f,fnevaluate' (i,newvenv) (l',t'))])) of
         v as VFUN(f,_) =>
          let val v' = set_valtype_env (i,newvenv) (v,t') in
           print("val "^newfun^" : "^
                   (descrtype(vtypeofv v'))^
                 " = "^
                   (stringof_value v')^"\n");
           setvvar (newvenv,newpos) v';
           local_venv := (!local_venv)@make_venv[(newfun,v')]
          end
       | v' => raise TypingBug("fun "^newfun^" does not evaluate to function!"^
                               "("^(stringof_value v')^")");
      true
     end
  | compile (VALDEF(i,e)) =
     let val _ = reset_vartypes()
         val newval = stringof_ident i
         val newvenv = ref(!local_venv)
         val e' = vnormalize newvenv e
         val t' = typesynthi ((i,newvenv),make_typesc(!newvenv)) e'
     in
      case vevaluate (i,newvenv) e' of
         v =>
          let val v' = set_valtype_env (i,newvenv) (v,t') in
           print("val "^newval^" : "^
                   (descrtype(vtypeofv v'))^
                 " = "^
                   (stringof_value v')^"\n");
           local_venv := (!local_venv)@make_venv[(newval,v')]
          end;
      true
     end
  | compile _ (* QUIT *) =
     false

end (* RSsmlComp *)
```

A.3.2 The Main Structure

```
(*
 * RSsmlMain.sml
 *
 * Raddstar Structured Meta Language
 * Expression Compiler -- Main Structure
 *
 * Copyright Martin Steeg, 1996 - 1999
 *)

signature RSSMLMAIN =
 sig
  val loop : unit -> unit
 end

structure RSsmlMain : RSSMLMAIN =
 struct
  open RSsml RSsmlComp

  structure RSsml_ParserLrVals =
        RSsml_ParserLrVals(structure Token = LrParser.Token);
  structure RSsml_ParserLex =
        RSsml_ParserLexFun(structure Tokens = RSsml_ParserLrVals.Tokens);
  structure RSsml_Parser =
        Join(structure LrParser = LrParser
             structure ParserData = RSsml_ParserLrVals.ParserData
             structure Lex = RSsml_ParserLex)

  val invoke = fn lexstream =>
    let val print_error = fn (s,i:int,_) =>
         say("Error, line " ^ (makestring i) ^ ", " ^ s ^ "\n")
    in
     RSsml_Parser.parse(0,lexstream,print_error,())
    end

  fun parse() =
       let val lexer = RSsml_Parser.makeLexer (fn _ => input_line std_in)
           val _ = RSsml_ParserLex.UserDeclarations.pos := 0
           val dummyEOF = RSsml_ParserLrVals.Tokens.EOF(0,0)
           val dummySEMI = RSsml_ParserLrVals.Tokens.SEMI(0,0)
           fun loop lexer =
                let val (result,lexer) = invoke lexer in
                 let val (nextToken,lexer) = RSsml_Parser.Stream.get lexer in
                  if RSsml_Parser.sameToken(nextToken,dummyEOF) then QUIT
                  else if RSsml_Parser.sameToken(nextToken,dummySEMI) then result
                  else loop lexer
                 end
                end
        in
         loop lexer
        end

  fun loop() =
       (psln();
        if (compile(parse())
             handle
                SUnbound s =>
                 (print("Unbound variable or identifier: "^s^"\n"); true)
              | TypingBug s =>
                 (print("Typing error: "^s^"\n"); true)
              | UNcaught id =>
                 (print("Uncaught match exception in function "^
                               (stringof_ident id)^"\n"); true)
              | RSsml_Parser.ParseError =>
                 (print("parse error\n"); true))
        then loop()
        else())
 end

val doit = RSsmlMain.loop
```

A.3.3 Application Scenario

```
RSsmlComp.sml:246.9-247.69 Warning: match nonexhaustive
          VB b => ...

RSsmlComp.sml:128.25-128.68 Warning: match nonexhaustive
          typevar (e,p) => ...

signature RSSMLCOMP =
  sig
    type stmt
    val compile : stmt -> bool
  end
structure RSsmlComp : RSSMLCOMP
[opening RSsmlMain.sml]
signature RSSMLMAIN = sig val loop : unit -> unit end
structure RSsmlMain : RSSMLMAIN
val doit = fn : unit -> unit
val it = () : unit
val it = () : unit
- doit();
RSsml> fun fac 0 = 1 | n = n*fac(n-1);
val fac : int -> int = <function>
RSsml> fun fib n = if n<=1 then 1 else fib(n-1)+fib(n-2) fi;
val fib : int -> int = <function>
RSsml> fun o(j,k) = fn x => j(k x);
val o : ('a -> 'b,'c -> 'a) -> 'c -> 'b = <function>
RSsml> o(fn (m,n) => fib(if m<n then fac m else fac n fi),fn j => j;
Error, line 0, syntax error

parse error
RSsml> o(fn (m,n) => fib(if m<n then fac m else fac n fi),fn j => j);
val it : (int,int) -> int = <function>
RSsml> o(3,4);
Typing error: ('a -> 'b,'c -> 'a) -> 'c -> 'b applied to (int,int)
RSsml> it(3,4);
val it : int = 13
RSsml> val k = o;
val k : ('a -> 'c,'b -> 'a) -> 'b -> 'c = <function>
RSsml> (k(fn (m,n) => fib(if m<n then fac m else fac n fi),fn j => j)) 2;
Typing error: (int,int) -> int applied to int
RSsml> (k(fn (m,n) => fib(if m<n then fac m else fac n fi),fn j => j)) (7,3);
val it : int = 13
RSsml> fib 13;
val it : int = 377
RSsml> val it = () : unit
-
```

Appendix B

Specification of the Schema Transformation and Optimization Rules

This Chapter shows the schema transformation and optimization rules which are (usually) preloaded by *load CSL from "..."* commands in the ".raddstar" initialization file of the RADD/raddstar system.

B.1 Rules for Hierarchical Transformation

B.1.1 Transformation Rule "h1"

```
add hierarchical transformation rule:
 when CC(s1,s2) is (1,1)
  and component s1 s2
 do
  group (s2,s1) (1,1)
```

B.1.2 Transformation Rule "h2"

```
add hierarchical transformation rule:
 when Reference s1->s2
  and exists s3:
   (component s1 s3 and tcomponent s3 s2)
 do not
  group (s2,s1)
```

B.2 Rules for Network Transformation

B.2.1 Transformation Rule "n1"

```
add network transformation rule:
 when CC(s1,s2) is (m,n)
   and component s1 s2
   and m >= 1
   and n <= !nMaxRepGrpSize
 do
  nest (s2,s1) list_t_
```

B.2.2 Transformation Rule "n2"

```
add network transformation rule:
 when Reference s1->s2
  and component s2 s1
 do
  separate s1 [s2]
```

B.3 Rules for Relational Transformation

B.3.1 Transformation Rule "r1"

```
add relational transformation rule:
 when CC(s1,s2) is (m,n)
  and component s1 s2
  and m >= 1
  and (n = 1 or (attrsize s1 (m,n)) < !rMaxRepGrpSize)
 do
  group (s2,s1) (m,n)
```

B.3.2 Transformation Rule "r2"

```
add relational transformation rule:
 when CC(s1,s2) is (0,1)
  and component s1 s2
  and emptySchema s1
 do
  group (s2,s1) (0,1)
```

B.4 Rules for Object-Relational Transformation

B.4.1 Transformation Rule "or1"

```
add objectrelational transformation rule:
 when CC(s1,s2) is (m,n)
  and component s1 s2
  and m >= 1
  and (n = 1 or (attrsize s1 (m,n)) < !rMaxRepGrpSize)
 do
  group (s2,s1) (m,n)
```

B.4.2 Transformation Rule "or2"

```
add objectrelational transformation rule:
 when CC(s1,s2) is (0,1)
  and component s1 s2
  and emptySchema s1
 do
  group (s2,s1) (0,1)
```

B.5 Rule for Object-Oriented Transformation

B.5.1 Transformation Rule "o1"

```
add objectoriented transformation rule:
 when CC(s1,s2) is (m,n)
  and component s1 s2
  and n <> ~1
 do
  nest (s2,s1) set_t_
```

B.6 Rules for Conceptual Schema Optimization

B.6.1 Optimization Rule "t1"

```
add conceptual optimization rule:
 when bottleneck(delete,s1) and bottleneck(delete,s2)
  and entity s1 and entity s2
  and exists s3,s4:
       ( (dcycle [s1,s3,s2,s4] or dcycle [s1,s4,s2,s3])
     and compatible [s3,s4] )
 do
  separate (group (s3,s4) (.,.)) [s4]
```

B.6.2 Optimization Rule "t2"

In contrast to the rules above, this is a special optimization rule for the Company schema. Refer also to [AAS97a, AAS97b].

```
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 (group (group (group (r4,r5) (1,1),r2) (.,.),r3) (.,.)) [r4,r5]
```

Appendix C

Development and Test Environment

C.1 Operating Systems and Development Tools

The version of RADD/raddstar that has been used to generate the screendumps of this work has been developed under a Linux Slackware 96 distribution, which was continously upgraded to more actual software packages, e.g. XF86Free-3.3.2, $\mathcal{K}$-Desktop Environment version 1.1, glibc6 (libc2), and the Linux kernel 2.2.13. To verify that the extensions of CML (see C.2.2) and eXene (see C.2.3) have the same behavior on different platforms, the system has been tested on a Sun Sparc 20 with 64 Mbytes of RAM. All tests (as well as the Y2K test, see C.4) were successful.

C.2 Standard-ML of New-Jersey (SML/NJ)

The SML/NJ Compiler (SML) was used as basis to implement the RADD/raddstar. Concurrent ML (CML) and eXene which are used in the RADD/raddstar are versions that have been extended. We have verified, that hese extensions have no impact on different behavior of the programming examples which are distributed with CML and eXene. That is, not using the extended features, CML and eXene behave as they were. Documentation about the extensions and how they can be used to implement your own X11-based applications, can be provided along with the source code of RADD/raddstar.

C.2.1 SML/NJ 0.93

The SML version used to compile the RADD/raddstar system is Standard-ML of New-Jersey (SML/NJ) version 0.93. Under Solaris the distribution that came originally from Princeton University (ftp://ftp.princeton.edu/pub/ml) is used. Under Linux a distribution of SML/NJ 0.93 which was found on diana.ibr.tu-bs.de is used. The Linux SML/NJ 0.93 version does only compile under early Linux distributions (such as Slackware 2.01,

which includes the 1.0.9 Linux kernel and the GNU C-Compiler (gcc) version 2.5.8 generating a.out executables). Therefore, the Linux SML executable (which is actually used) is in a.out format, but not in ELF. However, the Year 2000 (Y2K) Tests (Section C.4) with this executable were successful.

C.2.2 CML 0.9.8

The Concurrent ML (CML) code has been extended in Winter 1994/95 to run multiple CML machines under the same SML simultaneously. The CML 0.9.8 as was, permits only one RunCML.doit() function to be started and after that the "shell" (or function) that called the RunCML.doit() is blocked. By the modifications that we made, the "shell" is not blocked after calling RunCML.doit(). However, to start another RunCML.doit() the application must call ContCML.doit() which re-initializes respectively re-creates the static variables of the CML kernel, firstly. Look at the following code, which is taken from the RADD/raddstar structure `XListener`, that runs the Listener and the Listener GUI.

```
fun listen' (debugFlags, options) = (
   XDebug.init debugFlags;
   (* MS/991107: the next command spawns the thread for the listener *)
   RunCML.doit (fn () => run_listener options, SOME 20);
   (* MS/991107: the next two commands spawn the thread for the gui and the csl_loop *)
   ContCML.doit(fn () => RE.say("Starting the RADD* Listener GUI ...\n"), NONE);
   RunCML.doit (fn () => run_listener_gui options, SOME 20);
   (* MS/991213: the next command signal that CML has been started, but reads
               must not be requested from the CSL-Shell of the GUI (xrim_inch) *)
   RE.Static.CmlIsRunning := true;
   RE.Static.CslShellGUI := false;
   (* reading the ".raddstar" startup file *)
   readStartup();
   (* Command line parameter handling *)
   interprete options true COMLINEPARMS
   (* (true/COMLINEPARMS) indicates that CML is running *))

 fun listen options = let open ScanDD.UserDeclarations
               in sayDev := TERMDOT; listen'([],options) end
```

C.2.3 eXene 0.4

The SML/NJ lib (smlnj-lib-0.2) include loads have been changed such that eXene, which is the graphical user interface of SML/NJ and based on CML, loads the Unix library functions which are implemented by `unix-env.sml` and `unix-path.sml` as well. The eXene basics has been extended, whereby *changeable* widgets and *changeable* boxes were implemented. Look at the following code which is also taken from the RADD/raddstar structure `XListener`, that runs the Listener and the Listener GUI.

```
val lnMsgChan = (channel() : string chan)
(* channel to display the messages sent to the listener in the listener GUI *)

fun run_listener options =
 let fun quit root = (delRoot root; RunCML.shutdown())
     fun listener() =
          let fun inLoop() = (* watching the in channel *)
               let val index = save_typing_env() in
                case sync(receive XRIM.xrim_inch) of
```

```
                  XRIM.XRI_RULE HELP => (send(lnMsgChan,"help"); help())
                | XRIM.XRI_RULE QUIT =>
                    (case !wroot of
                        SOME root => (send(lnMsgChan,"quit"); quit root)
                      | _ => ())
                | XRIM.XRI_RULE r =>
                   ((interprete options true r;
                     case r of
                        LOADDD fnam =>
                         (send(lnMsgChan,"load DD from \""^fnam^"\"");
                          print"send(XRIM.xrim_ctlch,XRIM.XRI_CHANGEBUTTONS...)\n";
                          send(XRIM.xrim_ctlch,XRIM.XRI_CHANGEBUTTONS);
                          print"sent.")
                      | LOADCSL fnam =>
                         (send(lnMsgChan,"load CSL from \""^fnam^"\""))
                      | LOADML fnam =>
                         (send(lnMsgChan,"load ML from \""^fnam^"\""))
                      | CSL _ =>
                         (send(lnMsgChan,"CSL command"))
                      | _ => restore_typing_env index)
                       handle ParseError =>
                               (send(lnMsgChan,"*** parse error ***");
                                restore_typing_env index;
                                RE.say "Parse Error\n"; RE.psln())
                            | (SUnbound s) =>
                               (send(lnMsgChan,"*** unbound variable or identifier ***");
                                restore_typing_env index;
                                RE.say("Unbound variable or identifier: "^s^"\n"); RE.psln())
                            | (TypingBug s) =>
                               (send(lnMsgChan,"*** typing bug ***");
                                restore_typing_env index;
                                RE.say("Typing error: "^s^"\n"); RE.psln()))
                | XRIM.XRI_MESSAGE s => RE.say s
                | XRIM.XRI_GETLINE l => l := RE.get_input_line (!RE.instrmr)
                | _ => ();
               inLoop()
              end
          in
              spawn inLoop;
            ()
          end
 in
       listener()
 end

fun run_listener_gui options = let
     val root = mkRoot (displayScreenHost options)
     val _ = wroot := SOME root
     val (lnmes,lncur,lnpos) = (ref "",ref "-",ref 1)
     val MessageFont = "7x13"

     fun messageBox root = let
       val msgLabel = Label.mkLabel root {
         label="Listener:",
         foregrnd=NONE,
         backgrnd=NONE,
         font=SOME MessageFont,
         align=HLeft
       }
       val _ = (lnmes := ""; lncur := "-"; lnpos := 1) (* initialize on call *)
       val msgText = Label.mkLabelC root {
         label=(!lnmes),
         curpos=(!lnpos),
         foregrnd=SOME(W.EXB.whiteOfScr (screenOf root)),
         backgrnd=NONE,
         cforegrnd=SOME(W.EXB.blackOfScr (screenOf root)),
         cbackgrnd=NONE,
         font=SOME MessageFont,
         align=HLeft
       }
       val setFn = fn () => Label.setLabelC msgText (!lnmes,!lnpos)
       val _ = setFn()

       fun msgCtlLoop() = (* watching the ctl channel *)
             (case sync(receive lnMsgChan) of
                 message =>
                     (lnmes := ("received ==>> "^message);
```

```
                lnpos := (String.size message)+1;
                setFn());
          msgCtlLoop())

  val layout = (Shape.fixSize (Box.widgetOf(Box.mkLayout root (Box.VtCenter[
                                Box.Glue {nat=5, min=5, max=NONE},
                                Box.HzCenter
                                 [Box.Glue {nat=6, min=6, max=NONE},
                                  Box.HzCenter [Box.WBox (Label.widgetOf msgLabel)],
                                  Box.Glue {nat=6, min=6, max=NONE},
                                  Box.WBox
                                   (Shape.fixSize (
                                     Box.widgetOf(Box.mkLayout root
                                                  (Box.WBox(Label.widgetOf msgText))),
                                     let open G in SIZE{wid=250,ht=18} end)),
                                  Box.Glue {nat=6, min=6, max=NONE}],
                                Box.Glue {nat=6, min=6, max=NONE}])),
                                let open G in SIZE{wid=660,ht=50} end))
in
  spawn msgCtlLoop;
  layout
end

val behBttnBox = Box.cWBox (mkBehBttns root options)

fun mk_gui () =
 (Shell.mkCShell
   (Box.widgetOfCB
    (Box.mkLayoutCB root
      (Box.cVtLeft[
        Box.cGlue {nat=6,min=6,max=NONE},
        Box.cBoxOfB (Box.WBox (messageBox root)),
        Box.cGlue {nat=6,min=6,max=NONE},
        (Box.cHzCenter
          [Box.cGlue {nat=3,min=3,max=NONE},
           Box.cVtLeft
           [(Box.cVtCenter
             ([Box.cHzCenter[behBttnBox],
               Box.cGlue {nat=2,min=2,max=NONE}]))],
           Box.cGlue {nat=3,min=3,max=NONE},
           Box.cBoxOfB(Box.VtCenter([
            Box.WBox (Shape.mkRigid (mkFileOpBttns root)),
            Box.Glue {nat=2,min=2,max=NONE},
            Box.WBox(Box.widgetOf(Box.mkLayout root
                    (Box.WBox (Shape.mkRigid (mkReviewBttns root))))),
            Box.VtLeft
             ([Box.Glue {nat=4,min=4,max=NONE},
               Box.WBox(Shape.mkRigid(mkTransfBttn root)),
               Box.Glue {nat=0,min=0,max=NONE}]
              @ [(slidersBox root)])
          ])),
          Box.cGlue {nat=3,min=3,max=NONE}])])),
     NONE,
     {win_name = SOME "RADD/raddstar Listener GUI",
      icon_name = SOME "RADD/raddstar Listener GUI"}))
in
  setOptions{ins=(TaExtension.getInsertionOption()),
             del=(TaExtension.getDeletionOption())};
  Shell.initC (mk_gui());
  setSliders root;
  RE.Static.initprompt();
  RE.set_prfun (fn s => (RE.output(!RE.outstrmr,s); RE.flush_out(!RE.outstrmr)));
  set_prfun (fn s => RE.say("\n"^s^" ..."));
  if not("-LOADDD" in_l options) then RE.psln() else();
  ()
end
```

C.2.4 Port to SML/NJ 1.10

A first try to port the modified versions of CML and eXene to the actual Standard-ML of New-Jersey (SML/NJ 1.10) failed. As soon as possible CML and eXene used here, as well as the RADD/raddstar will be ported to SML/NJ 1.10 (or later).

C.3 Postgres

The programming example in Section 17 has been developed using a modified version of the *postgresql-6.5-beta1* distribution and a shared library, implemented by the author.

1. Modified PGSQL-Parser and Data Dictionary.
   The parser of Postgres was modified such that the language allows to define attribute types $char(n_1)[n_2]$ and $varchar(n_1)[n_2]$ where n_1 and n_2 are numbers, respectively. Accordingly, we have inserted data dictionary tuples to consider these types, which are not supported by the *postgresql-6.5-beta1* distribution as is. I.e., we have extended the Postgres type system such that the char and varchar types have been made avalaible as base types for arrays.
2. Shared library T_EName.so.
   We have programmed a module T_EName.c and generated the shared library T_EName.so, which implements the Employee-Name type
   - the size of EName is 34 byte:
     Firstnames(which is a pointer to an array of char[15] fields, => 4 bytes)
     Lastname which is char[20], and Title which is char[10]
   - the input and output functions are:
     EName* ename_in(char* str) and char* ename_out(EName* name)

C.4 Year 2000 (Y2K) Tests

Unix operating systems store the time in a pointer to a signed long (type `time_t *`), beginning at the 1st of 1970 0:00 Greenwich Mean Time (GMT) which equals zero. Unix operating systems have normally no problem with the Y2K, besides it is an older release of the operating system (such as Sun Solaris 2.5.1, or earlier), and applications use functions to convert the Unix internal time format to the "readable" time format; i.e., something like "YYYY:MM:DD:hh:mm:ss" (which is stored by means of the C-type `struct tm *`).

The critical time conversion functions are the following two functions, which are not correctly implemented in the C-library (libc.so) of Solaris 2.5.1, or ealier:

- *localtime(),* which converts from the Unix time format (`time_t *`) to the "readable" format (`struct tm *`), and
- *mktime(),* which does the opposite (`struct tm *` converted to `time_t`).

Under Solaris 2.5.1, for a date in 1999 localtime returns the year 99. If the Unix time is over the Year 2000, then localtime returns the year - 1900; that is, 100 for the Year 2000, 101 for the Year 2001, and so forth. The mktime function works incorrectly if the year is

greater than 99. Then, it returns a -1, which represents the 31st of December 1969 23:59 59 seconds.

As far as I have inspected the source code of Standard ML of New-Jersey Version 0.93, which was used to implement the RADD/raddstar, it does not make any time conversion, but maintains the Unix internal time only. Besides that, the RADD/raddstar system was tested on a 686-PC with a 1997 Bios running a 1996 Linux distribution (the SML compiler used was still a.out format, built on a 1994 Linux distribution), such that the time was set to and over the 1st of January 2000. I compiled and tested the system completely with the date set over 2000 on both platforms, and I could not recognize any difference to the behavior when the machine (686 resp. Sparc) was running with the date set 1999.

The date and time conversion functions which are used in the RADD/raddstar, have been implemented especially for that purpose and are working correctly. (Refer to the example shown in Section 7.2.2.)

Appendix D

Catalog of Terms and Abbreviations

1NF. First normal form → Normalization. Refer to Section 2.3.2.1.

2NF. Second normal form → Normalization. Refer to Section 2.3.2.2.

3NF. Third normal form → Normalization. Refer to Section 2.3.2.3.

4NF. Fourth normal form → Normalization. Refer to Section 2.3.3.2.

5NF. Fifth normal form → Normalization. Refer to Section 2.3.3.4.

AD. Afunctional Dependency → Integrity Constraint.

Additional Requirements. Refer to Section 8.2.

Algebraic Specification. Refer to Section 6.1.

Application Programming. Refer to Section 7.3.

Attribute. A field of a Structure that carries a value of a certain type (integer, string, etc.). Refer to Section 2.1.

BCNF. Boyce-Codd normal form → Normalization. Refer to Section 2.3.2.4.

Balancing Parameter. Refer to Section 5.3.1.3.

Basic Schema Transformation Operation. Refer to Section 5.2.2.

Behavior Option. Specification for control of the behavior on database modifications by the DBMS *(restrict, cascade, set null, set default)*. Refer to Section 5.3.2.

Behavior Specification. Specification for control of the behavior on database modifications by the DBMS. For simplication, RADD/raddstar supports → Behavior Options. Refer to Section 5.3.2.

Booch Method. An object-oriented design method based on decomposition, abstraction, and hierarchy. Refer to Section 3.2.1.

Btree. B-tree access and storage method for databases ("clustering index"). See Sections 2.3.5.2, 5.3.1.1, 5.3.1.2, and 5.3.1.3.

CC. Cardinality Constraint → Integrity Constraint.

CSL. Conceptual Specification Language. A database design specification and programming language with functional → SML-like extensions. CSL is the command-line user interface of the RADD/raddstar. Refer to Chapter 7.

Coad/Yourdon Method. An object-oriented database design method. Refer to Section 3.2.3.

Control Structures. Refer to Section 7.3.

Constraint → Integrity Constraint.

Correct Database States. Refer to Section 5.3.2.3.

Cost Evaluation. Refer to Section 5.3.3.

Cost Parameter Function. Refer to Section 5.3.1.2.

"D" Database Model. An specification of concepts for new-generation databases combining concepts of the → Relational Data Model and Object-oriented Data Models. Refer to Section 3.3.2.4.

DBA → Database Administrator.

DBMS → Database Management System.

Database. A collection of data which is typically stored structurally in the form of tables– in contrast to data stored by a → File System. (See Chapter 2.)

Database Administrator. Person who is resposible for the administrative tasks of a → Database, such as creating the database, starting and shutting down database services, creating and enlarging tablespaces, rollback segments, and logfiles, making backups, monitoring transactions, etc.

Database Design. Designing the → Structures and → Relationships of a → Database for implementations under a → DBMS.

Database Design Repair. Refer to Section 4.2.1.

Database Management System. A collection of programs to create and maintain a → Database. The programs of a Database Management System are used as a general purpose software system for specifying, constructing, and maintaining the database for the various applications who need to access and modify the data.

Database Trigger. An action (A) that is automatically performed by the → DBMS as soon as a special condition (C) occurs after happening of an event (E). All modeling, specification and implementation techniques for Database Triggers use the *ECA* model or some variant of it. Refer to Section 1.3.3.

DBtree. Dense B-tree access and storage method for databases ("non-clustering index"). See Sections 2.3.5.2, 5.3.1.1, 5.3.1.2, and 5.3.1.3.

Dynamic Constraint. Special kind of → Integrity Constraint having dynamic behavior. Refer to Section 3.3.2.1.

ECA. Event-condition-action → Database Trigger.

EC^{2}A. Event-constraint-condition-action. The → Transaction Extension and evaluation model of the RADD/raddstar.

ED. Exclusion Dependency → Integrity Constraint.

EERM → Extended Entity-Relationship-Model.

ERM → Entity-Relationship-Model.

Extended Entity-Relationship-Model. All extensions of Chen's Entity-Relationship-Model of 1976 are today denoted as extended Entity-Relationship-Model. There are several kinds of extensions, e.g. record-typed, list-typed or set-typed attributes or the annotation of new kinds of integrity constraints. (See Chapter 3.)

Entity-Relationship-Model. A methology for specifying a database by means of entities (entity is the type of an object of the real world) and relationsships (relationship is the type of an association between objects of the real world). Entities and Relationships have attributes desribing the properties of the according object, e.g. the name of a Person or the date a Person marries another. It is also possible to define integrity constraints for entities and relationships, e.g. the Person's name attribute is called a key if it can be used to identify a Person uniquely in the Company (that is, no two persons in the database have the same name). The Entity-Relationship-Model originally was introduced by Peter Chen in 1976. (See Section 2.1.)

Error Prevention Properties. Refer to Section 5.1.1.

FD. Functional Dependency → Integrity Constraint.

File System. A collection of directories and files. Directories contain directories and files. First generation databases (as well as unfortunately some "object-oriented" databases) were mapped with the help of file systems (i.e., to directories and files).

Fitness Evaluation. Refer to Section 5.3.3.

Functional Specifications. Refer to Section 7.2.

GUI. Graphical User Interface.

GemStone. Object-oriented DBMS. Refer to Section 3.3.2.3.

HERM → Higher-order Entity-Relationship-Model.

Higher-order Entity-Relationship-Model. An extended Entity-Relationship-Model which adds relationsships between relationsships to the ERM methology. The Higher-order Entity-Relationship-Model has been proposed by Bernhard Thalheim in 1989. (See Section 3.3.3.1.)

HTML. Hypertext Markup Language. A programming language for the design of Web sites.

Hash. Hash(ing) access and storage method for databases. See Sections 2.3.5.2, 5.3.1.1, 5.3.1.2, and 5.3.1.3.

Heap. Heap access and storage method for databases. See Sections 2.3.5.2, 5.3.1.1, 5.3.1.2, and 5.3.1.3.

Hierarchical Data Model. A data model based on hierarchical ordering of the data. Refer to Section 2.2.

Hierarchical Model. → Hierarchical Data Model.

ID. Inclusion Dependency → Integrity Constraint.

IFO. A formal semantics database model. Refer to Section 3.1.3.

ISAM. Index-Sequential Access Method. See Sections 2.3.5.2, 5.3.1.1, 5.3.1.2, and 5.3.1.3.

Informix. A Relational DBMS. Refer to Section 2.3.5.

Ingres. A Relational DBMS. Refer to Section 2.3.5.

Insertion Option. → Behavior Specification for the control of insertions in network databases. Refer to Section 2.2.5.

Integrity Constraint. A presupposition necessary for consistency of the data. E.g., a key attribute identifies an object uniquely in a set of objects. Or, the Person's name is used to identify each Employee uniquely (in a small Company): There can be no two Persons (Person records in the database) having the same name. Or, an inclusion dependency specifes that the values of an attribute must exists as values of another attribute (of probably, a different relation). (For detailed explanation of Integrity Constraints, refer to Section 2.3.1.)

Integrity Maintenance. Techniques to ascertain the integrity of data in the database. This can be done by installing the → Relation Schema such that → Integrity Constraints can not be violated by the database operations (refer to Section 2.2.5, 2.3.2 and 2.3.3) or by invoking subsequent → Repairing Actions which migrate temporarily inconsistent databases to consistent ones (refer to Section 3.3.2.3).

Kett-Entity. Record type to represent many-to-many relationships in the network data model. Refer to Section 2.2.4.

Key → Key-Attribute.

Key-Attribute → Integrity Constraint.

LList. Linked List access and storage method for databases. See Sections 2.2.4, 2.3.5.2, 5.3.1.1, 5.3.1.2, and 5.3.1.3.

Lock Tuning. → Tuning the operating of the DBMS according the locks it is setting on the data sets. Refer to Section 4.1.2.

ML. Meta Language. A functional programming language. Refer to Section 6.2.

$(NF)^2$. Non-first normal form → Normalization. Refer to Section 2.3.2.1.

NIAM. Nijssens Information Analysis Method. A database modeling and specification technique considering database types (lexical object types, LOTs → Attribute; non-lexical object types, NOLOTs → Structure), facts between the LOTs and NOLOTs (→ Relationships), and populations of the LOTs, NOLOTs, and facts.

Nested Relational Model. Extended → Relational Data Model. Refer to Section 3.3.2.2.

Network Data Model. A data model based on network-like connected data. Refer to Section 2.2.

Network Model. → Network Data Model.

Normalization. A technique to reduce overhead in a set of → Relation Schemata by generating a new set of relation schemata for the purpose to reduce the cost of → Integrity Maintenance. Refer to Section 2.3.2 and 2.3.3.

O_2. Object-oriented DBMS. Refer to Section 3.3.2.3.

OMT → Object Modeling Technique.

ONF. Optimal Normal Form → Normalization. Refer to Section 4.1.1.4.

ORM → Object Role Model.

Object Role Model. A variant of the → NIAM. Refer to Section 3.3.1.

Object Modeling Technique. An object-oriented database design model. Refer to Section 3.2.2.

Optimization. Improving something that it behaves optimally.

Oracle7. A Relational DBMS. Refer to Section 2.3.5.

Oracle8. An Object-Relational DBMS. Refer to Section 3.3.2.3.

Ontos. Object-oriented DBMS. Refer to Section 3.3.2.3.

PAD. Path Afunctional Dependency → Path, → Integrity Constraint.

PCC. Path Cardinality Constraint → Path, → Integrity Constraint.

PED. Path Exclusion Dependency → Path, → Integrity Constraint.

PFD. Path Functional Dependency → Path, → Integrity Constraint.

PID. Path Inclusion Dependency → Path, → Integrity Constraint.

PREF. Path Reference → Path, → Integrity Constraint.

Path. Paths can be looked at as collections of structures (attribute, entity, relationship, cluster) that are connected some way. Typically a path is a join between two or more → Structures ($S_1 \bowtie S_2 \ldots$) such that this is used to specify a more general type of → Integrity Constraint.

Path Dependency. → Integrity Constraint, → PAD, PCC, PED, PFD, PID, PREF. (Refer to Section 6.3.1.2.)

plausibility Function. Refer to Section 5.3.2.4.

plau_card Function. Refer to Section 5.3.2.4.

Population Information. Refer to Section 7.1.1.

Postgres. Object-oriented DBMS (Postgres should be classified as an rather Object-Relational DBMS). Refer to Section 3.3.2.3.

Preceder. Refer to Section 5.2.1.1.

RADD* Data Model. The data model representing the types of the internally processed data of the RADD/raddstar. Refer to Section 6.3.

REF. Reference → Integrity Constraint.

RTS. Rule-Triggering-System. See → Database Trigger.

Relation. A set of tuples having all the same → Structure. See → Relation Schema.

Relation Schema. A set of → Attributes used to describe the types and labels of the fields of a tuple (record) in a set of tuples (→ Relation, all tuples must be of the same type.)

Relational Data Model. A data model based on sets of tuples describing relations between the values of the tuples (→ Relation Schemata). Refer to Section 2.3.

Relational Model → Relational Data Model.

Requirements' Specification. Refer to Section 8.2.

Repairing Actions. Actions to repair temporarily inconsistent database states by means of → Database Triggers. See also → Integrity Maintenance.

Retention Option. → Behavior Specification for the control of deletions in network databases. Refer to Section 2.2.5.

SDM. Semantic Data Model. Refer to Section 3.1.2.

SML. Standard ML of New-Jersey. Refer to Section 6.2.

SQL → Structured Query Language.

SQL-2 → SQL-92.

SQL-3 → SQL-99.

SQL-92. The SQL specification of 1992, also known as SQL-2.

SQL-99. The SQL specification of 1999/2000, also known as SQL-3. SQL-3 is supported (e.g. Oracle8) or will be supported by the new-generation DBMSs. Refer to Section 3.3.2.3.

Schema Reviewing. Refer to Section 8.3.

Schema Optimization. Refer to Section 8.3.

Schema Transformation Operation. Refer to Section 5.2.

Schema Transformations. Refer to Section 7.1.2.

Structure. A structure with labeled fields (→ Attribute) used to hold tuple data. In this work, the term Structure is used to describe what is denoted record type in the → Hierarchical Data Model and → Network Data Model, and what is denoted → Relation Schema in the → Relational Data Model. Refer to Section 6.3.

Structured Query Language. Programming language for structural definition and maintenance (retrievals, insertions, deletions etc.) of a database. SQL today is used as the standard programming interface of almost all commercial DBMSs.

Sybase. A Relational DBMS. Refer to Section 2.3.5.

System Architecture. Refer to Section 8.1.

Transaction. A database operation or sequence of database operations (insertions, deletions, updates, retrievals) that are only valid as a whole, and so, generate a new database state only as a whole. If one of the operations in the sequence of database operations fails, all other operations, even if they or some of them succeeded, are rejected by the → DBMS.

Transaction Chopping. A → Tuning technique by means of breaking → Transactions into smaller parts. Refer to Section 4.1.2.

Transaction Extension. Extending database operations (→ Transactions) such that new transactions are generated containing all necessary actions for deriving a consistent database state (→ Integrity Maintenance). Refer to Section 5.3.2.

Transaction Graph Mapping. Refer to Section 5.3.3.

Trigger → Database Trigger.

Tuning. Improving something such that it behaves more well. In the context of database tuning, Tuning and → Optimization are often used as synonyms.

Typing Rules. Rules for the derivation of types from patterns in sentences of a programming language such as → SML or → CSL. Refer to Section 6.2.2.2.

Union. A variant (or "union") type. In this work, Union is used to describe the structure of a set of tuples having a variant type. In contrast, → Structure describes a non-variant type.

VPCR. Virtual Parent-Child Record. Refer to Section 2.2.2.

WiSS. Wisconsin Storage System. See Sections 5.3.1.1, 5.3.1.2, 5.3.1.3, and 3.3.2.3.

Appendix E

Bibliography

E.1 Data Models and Database Management Systems

[ABD+89] M.P. Atkinson, F. Bancilhon, D.J. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system Manifesto. In *[KNN89]*, pages 40 – 57, 1989. 2

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM ToDS*, 12(4):525 – 565, December 1987. 1.2, 3.1, 3.1.3, 9

[AHW95] A. Aiken, J.M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM ToDS*, 20(1):3 – 41, March 1995. 1.3.3, 3.3.2.1

[ASK94] ASK. "OpenIngres" heißt der neue ASK-Sprößling. *Datenbank Fokus*, pages 30 – 33, 03/04 1994. (in German). 17

[BO96] P. Bunemann and Atsushi Ohori. Polymorphism and Type Inference in Database Programming. *ACM ToDS*, 21(1):30 – 76, March 1996. 6.2.2, 7

[Bom94] P.van Bommel. Experiences with EDO: An Evolutionary Database Optimizer. In *Data & Knowledge Engineering*, 13, pages 243 – 263, 1994. 1.5

[Bom95] P.van Bommel. *Database Optimization - An Evolutionary Approach*. PhD thesis, Katholieke Universiteit Nijmegem, 1995. 1.5, 3.3.1, 4.1.1.2, 4.1.1.4

[Boo94] G. Booch. *Object-oriented Analysis and Design*. Addison-Wesley, 2nd edition, 1994. 3.2.1

[BWL94] P.van Bommel, ThP.v.d. Weide, and C.B. Lucasius. Genetic algorithms for optimal logical database design. In *Information and Software Technologie*, number 12 in 36, pages 725 – 732, 1994. 1.5

[Cam94] L. Campbell. Adding a New Dimension to Flat Conceptual Modeling. In *First International Conference on Object-Role Modelling ORM-1*, July 1994. 1.5

[Car94] R.G.G. Cartell, editor. *The Object Database Standard: ODMG-93*. Morgan-Kaufmann, 1994. 1.3.4, 2, 11

[CBC93] S. Choenni, H. Blanken, and T. Chang. Index selection in relational databases. *IEEE Computing Surveys*, pages 491 – 496, 1993. 5.3.1, 5.4

[CDKK85] H. Chou, D. DeWitt, R. Katz, and T. Klug. Design and Implementation of the Wisconsin Storage System (WiSS). In *Software Practice and Experience*, 15(10), 1985. 1.5, 15, 5.3.1.2

[CFPT94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic Generation of Production Rules for Integrity Maintenance. *ACM ToDS*, 19(3):367 – 422, September 1994. 5.4

[CG93] P. Corrigan and M. Gurry. *ORACLE Performance Tuning*. O'Reilly & Associates, Inc., 1993. 4.1.1.1, 4.2.1

[Che76] P.P.S. Chen. The Entity-Relationship Model: Towards a Unified View of Data. *ACM ToDS*, 1(1):1 – 36, March 1976. 1.2, 2

[Cod70] E.F. Codd. A Relational Model for Large Shared Data Banks. *Comm. of the ACM*, 13(6):197 – 204, 1970. 1.3.1, 2, 2.3, 4.1.1

[Cod79] E.F. Codd. Extending the database relational model to capture more meaning. *ACM ToDS*, 4:397 – 434, Dec. 1979. 1.3.1, 2.3

[Con96] The ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *ACM Sigmod Record*, 25(3):40 – 49, September 1996. 2

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. 16th Conf. on Very Large Data Bases*, pages 566 – 577. Morgan Kaufman Publishers, 1990. 3.3.2.1

[CW94] S. Chakravarty and J. Widom, editors. *Research Issues in Data Engineering: Active Databases*. Proc., Houston, 1994. 1.3.3

[Dat92] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 4th edition, 1992. 7.1

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM Sigmod Record*, 17(1), March 1988. 1.3.3, 3.3.2.1

[DD95] H. Darwen and C.J. Date. The Third Manifesto. *ACM Sigmod Record*, 24(1):39 – 49, March 1995. 2, 3.3.2.4

[DTG71] DTG. Committee on Data System Languages (Codasyl) – *Database Task Group Report*. ACM Press, New York, April 1971. 2.2.7

[DTG79] DTG. Report of the database description language committee. *Information Systems*, 3:247 – 320, 1979. 2.2.7

[Dun98] J. Dunham. *Database Performance Tuning Handbook*. McGraw-Hill, 1998. 1.5, 1.5, 4.1

[EG97] D.W. Embley and R.C. Goldstein, editors. *Conceptual Modeling – ER'97, 16th International Conference on Conceptual Modeling*, San Diego, November 1997. E.4

[EN89] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989. 2, 2, III

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 2nd edition, 1994. 2.2.3, 2.2.7

[Eve88] G. Everest. *Database Management*. McGraw-Hill, 1988. 4.1.1.4

[Fag77] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM ToDS*, 2(3):262 – 278, 1977. 2.3.3.1, 2

[FSS88] J. Fladeiro, A. Sernadas, and C. Sernadas. Knowledgebases as structured theories. In *Proc. 8th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 338 of *LNCS*, pages 469 – 486. Springer-Verlag, 1988. 6.1

[GCS+97] Rick Greenwald, John Davidson Conley III, Steve Shiftett, Joseph Duer, Jeffrey Dwight, Simeon Greene, Alexander Newman, and Scott Williams. *Using Oracle Web Application Server 3*. Que, December 1997. 8

[Gil91] Dov Gilor. *SQL/DS Performance*. John Wiley & Sons, Inc., 1991. 4.2.1, 5.3.1

[Gün95] M. Günther. SQL-Dilemma. *iX*, pages 160 – 164, June 1995. 5.1

[Hai89] J.-L. Hainaut. A generic entity-relationship model. In *Information System Concepts: An In-depth Analysis*, Amsterdam, The Netherlands, 1989. Elsevier Science Publishers. 1.5

[Hal90] T.A. Halpin. Conceptual schema optimization. *Australian Computer Science Communications*, 12(1):136 – 145, 1990. 1.5

[Hal91] T.A. Halpin. A fact-oriented approach to schema transformations. In *MFDBS'91*, volume 495, pages 342 – 356. LNCS, 1991. 1.5

[Han95] M.S. Hanna. A Close Look at the IFO Data Model. *ACM Sigmod Record*, 24(1):21 – 26, March 1995. 3

[Heu92] A. Heuer. *Objekt-orientierte Datenbanken*. Addison-Wesley, 1992. (in German). 7

[HM81] M.M. Hammer and D.J. McLeod. Database Description with SDM: A Semantic Database Model. *ACM ToDS*, 6(3):351 – 386, September 1981. 1.2, 3.1, 3.1.2

[ILR95] J. Iirari, K. Lyytinen, and M. Rossi, editors. *Proc. 7th Int. Conf. on Advanced Information System Engineering, CAiSE'95*, number 932 in LNCS, Jyväskÿlä, Finland, June 14 - 16 1995. E.4

[Kim95] Won Kim, editor. *MODERN DATABASE SYSTEMS – The Object Model, Interoperability and Beyond*. ACM Press, 1995. 7

[KL78] Dan Kapp and Joseph F. Leben. *IMS Programming Techniques – A Guide to Using DL/1*. Van Nostrand Reinhold Company, 1978. 2.2.3

[KNN89] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, Kyoto, December 1989. Elsevier. E.1

[KP76] L. Kershberg and J.E.S. Pacheco. A Functional Database Model. Technical report, Pontificia Universidade Catolica do Fio de Janeiro, Brazil, February 1976. 1.2, 3.1.1

[KS91] H.F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, 1991. 1.5, 5.4, 47

[LN88] C.M.R. Leung and G.M. Nijssen. Relation Database Design using the NIAM Conceptual Schema. *Information Systems*, 13(2):219 – 227, 1988. 4.1.1.4

[MNE96] Way Yin Mok, Yiu-Kai Ng, and David W. Embley. A normalform for precisely characterising redundancy in nested relations. *ACM ToDS*, 21(1):77 – 106, March 1996. 3.3.2.2

[Nij77] G.M. Nijssen. Current Issues in Conceptual Schema Concepts. In *Architecture and Models in Data Base Systems*, pages 31 – 65. North-Holland, 1977. 1.2, 3.3.1

[Oll78] T.William Olle. *The Codasyl Approach to Data Base Management*. John Wiley & Sons, Inc., 1978. 2.2.7

[Oracle8] Oracle. *Oracle8*, 1998. (Online HTML Documentation for Oracle 8.05 DBMS). 2

[PBGG89] J. Paredaens, P.D̃e Bra, M. Gyssens, and D.Ṽan Gucht. *The Structure of the Relational Database Model*. EATCS Monographs on Theorethical Computer Science. Springer-Verlag, Berlin, 1989. 3.3.2.2, 6.3.2

[PCO95] J.A. Pastor-Collado and A. Olivé. Supporting Transaction Design in Conceptual Modelling of Information Systems. In *LNCS 932*, 1995. 6.1

[PLSQL95] Oracle. *Oracle – Server SQL Reference. Release 7.2*, April 1995. (Edited by Brian Linden). 4, 4.2.1

[PLSQL98] Oracle. *PL/SQL User's Guide and Reference. Release 8.0*, June 1998. (Online HTML Documentation). 8

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991. I, 3.2.2

[RK87] M.A. Roth and H.F. Korth. The design of non-1NF relational databases into nested normal form. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 16(2), May 1987. 3.3.2.2

[RKS88] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM ToDS*, 13(4):389 – 417, December 1988. 3.3.2.2

[RM92] A. Raihä and H. Manila. *The Design of Relational Databases*. Addison-Wesley, 1992. 4.1.1.4, 7.1

[RR94] A. Rosenthal and D. Reiner. Tools and Transformations – Rigorous and Otherwise – for Practical Database Design. *ACM ToDS*, 19(2):167 – 211, June 1994. 5.4

[Sha92] D.E. Shasha. *Database Tuning – A Principled Approach*. Prentice Hall, 1992. 1.3, 1.5, 1.5, II, 4, 4.1, 4.1.1.2, 4.1.2.2, 4.2.1, 4.3, 47

[Shi81] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM ToDS*, 6(1):140 – 173, March 1981. Also found in [ZM90]. 1.2, 3.1, 3.1.1

[SK77] E.H. Sibley and L. Kershberg. Data architecture and data model considerations. pages 85 – 96, June 1977. 3.1.1

[SST94] K.-D. Schewe, D. Stemple, and B. Thalheim. Higher-level genericity in object-oriented databases. In S. Chakravarty and P. Sadanandan, editors, *Proc. COMAD*, Bangalore, 1994. 1.3.3, 3.3.2.1, 5.4

[ST92] K.-D. Schewe and B. Thalheim. Computing consistent transactions. Technical Report CS-08-92, University of Rostock, December 1992. 3.3.3.1, 5.3, 5.4

[ST94a] K.-D. Schewe and B. Thalheim. Achieving Consistency in Active Databases. In S. Chakravarty and J. Widom, editors, *Proc. RIDE-ADC*, Houston, 1994. 1.3.3, 3.3.2.1, 5.4

[ST94b] K.-D. Schewe and B. Thalheim. A computational approach to consistency enforcement. In *Proc. of the Int. Workshop on Combining of Declarative and Object-Oriented Databases*, pages 111 – 124, 1994. 3.3.3.1, 5.3, 5.4

[ST98] K.-D. Schewe and B. Thalheim. On the Strength of Rule Triggering Systems for Integrity Maintenance. In *Australian Database Conference*, January 1998. 5.4

[Su85] S.S. Su. *Processing-Requirement Modeling and Its Application in Logical Database Design*, pages 151 – 173. 1985. 1.5, 4.1.1.2

[Syb93] Inc. Sybase. *Transact-SQL User's Guide*, September 1993. 4, 4.2.1

[TDF86] T. Teorey, Yang D., and J.P. Fry. A Logical Design Methology for Relational Databases Using the Extended Entity-relationship Model. *ACM Computing Surveys*, 18(2):197 – 222, June 1986. 4.1.1

[Tha89] B. Thalheim. The Higher-order Entity-Relationship-Model and $(DB)^2$. In *LNCS*, volume 364, pages 382 – 397. Springer-Verlag, 1989. 3.3.3.1

[Tha91] B. Thalheim. *Dependencies in Relational Databases*. Teubner Verlag, Leipzig, 1991. 5, 6.3.2

[Tha96] Bernhard Thalheim, editor. *Conceptual Modeling – ER'96, 16th International Conference on Conceptual Modeling*, Cottbus (Germany), October 1996. E.4

[Tha97] B. Thalheim. *Fundamentals of Entity-Relationship Modeling*. Springer-Verlag, 1997. 3.3.3.1, 5, 3.3.3.2

[Ull82] J.D. Ullmann. *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, 1982. 4.1.1.4

[Ull88a] J.D. Ullmann. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, Maryland, 2nd edition, 1988. 2, 2.2.4

[Ull88b] J.D. Ullmann. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Maryland, 2nd edition, 1988. 7.1

[VB82] G.M.A Verheyen and J.van Bekkum. *NIAM: An Information Analysis Method*, pages 1– 53. Informatica B.V. and Control Data. Nijssen Advieusbureau, The Netherlands, December 1982. 1.2

[Vos87] G. Vossen. *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. Addison-Wesley, 1987. (in German). 7.1

[Wie87] G. Wiederhold. *File Organization for Database Design*. McGraw-Hill, 1987. 1.5, 5.3.1, 5.4, 47

[Win85] J.J.V.R. Wintraekken. *Informatie Analyse volgens NIAM*. North-Holland, 1985. (in Dutch). 4.1.1.4

[X3.92] ANSI X3.135-1992. *American National Standard for Information Systems – Database Languages – SQL*. ANSI, November 1992. 4

[ZM90] StanleyB. Zdonik and David Maier. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, San Mateo, California, 1990. E.1

E.2 Formal Database Specification Approaches

[AE91] S. Antoy and C. Egyhazy. Modelling Databases using Algebraic Specification. (Preprint), 1991. 6.1

[Bro87] M. Broy. Specification of a Railway System. Technical Report MIP-8715, Universität Passau, 1987. (in German). 6.1

[EKW79] H. Ehrig, H.-J. Kreowski, and H. Weber. Neue Aspekte algebraischer Spezifikationsschemata für Datenbanksysteme. In *IFB 21 Proc. Workshop Formale Modelle für Informationssysteme*, pages 181 – 198. Springer-Verlag, 1979. (in German). 6.1

[EW78] H. Ehrig and H. Weber. Algebraic specification schemes for data base systems. In *Proc. 4tn Conf. on Very Large Databases*, Berlin, 1978. 6.1

[Gog93] M. Gogolla. Spezifikation von Datenbanken mit Troll Light. Technical report, Universität Braunschweig, 1993. (in German). 6.1

E.3 Functional Languages

[Bru62] N.De Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 1962. 6.2.2

[Fai85] J. Fairbairn. Design and implementation of a simple typed language based on the lambda-calculus. Technical Report 75, University of Cambridge, 1985. 6.2.2

[GMM+78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadworth. A metalanguage for interactive proofs in lcf. In *Symp. on Princ. Prog. Lang.*, pages 119 – 130. ACM, 1978. 6.2

[GWM+91] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannad. Introducing OBJ*. Technical report, Michaelmas, DRAFT: August 14, 1991. 6.1

[Mac62] J. MacCarthy. *Lisp 1.5 Programmer's Manual*. Cambridge, Mass., 1962. 6.2

[Mau93] M. Mauny. Functional programming using Caml Light. Technical report, INRIA, Paris, 1993. 6.1, 6.2.1.1, 5, 6.2.2.2, 6.2.2.2, 6.2.2.2, 6.2.2.3, 7.3.2

[Mau95] M. Mauny. Functional programming using Caml Light. Technical report, INRIA, Paris, January 1995. (Caml Light version 0.74). 6.2.2.3

[Mil78] R. Milner. A proposal for type polymorphism in programming. *Computing Systems*, 17:348 – 375, 1978. 6.2.2

[Mil87] R. Milner. A Proposal for Standard ML. In *Proc. ACM Conference on Lisp and Functional Programming*, 1987. 6.2, 7.3.2

[Pau90] L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361 – 386. Academic Press, 1990. 31

[Rep91] John H. Reppy. Cml: A higher-order concurrent language. In *Proc. SIGPLAN'91 Conf. on Programming Language. Design and Implementation*, pages 293 – 305, June 1991. 4

[Rep93] John H. Reppy. *CML: A Higher-order Concurrent Language*. Cornell University, February 15, 1993. 4

[Tur76] D. Turner. SASL language manual. Technical report, St. Andrews University, 1976. 6.2

[Tur85] D. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1 – 16, 1985. 6.2

E.4 The RADD Project

[AAB+95] 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 *[ILR95]*, 1995. 3.3.3.2, 8.1

[AAS97a] M. Albrecht, M. Altus, and M. Steeg. Conceptual Data Modeling, Implementation Prototyping, Transformation, and Application Design – An Animating Approach using RADD. In *ADBIS – Conference on Advanced Techniques in Databases and Information Systems*, St. Petersburg (Russia), September 1997. 3.3.3.2, 8.1, B.6.2

[AAS97b] M. Albrecht, M. Altus, and M. Steeg. Modeling of Behavior of Databases: A Transformational Approach using RADD. In *[EG97]*, November 1997. 3.3.3.2, 8.1, B.6.2

[Alb94] M. Albrecht. Semantikakquisition im Reverse-Engineering. Technical Report I - 4/1994, TU Cottbus, Institut für Informatik, June 1994. (in German). 2.3.4.3, 4, 3.3.3.2, 8.1

[Bac91] P. Bachmann. Antrag zur Gewährung einer Sachbeihilfe durch die Deutsche Forschungsgemeinschaft zur Thematik Modellierung von Struktur, Semantik und Verhalten von Datenbanken – operationale Spezifikation –, September 1991. 6

[BDT94] E. Buchholz, A. Düsterhöft, and B. Thalheim. Exploiting Knowledge Gained from Natural Language for EER Database Design. Technical Report I - 10/1994, Cottbus Technical University, 1994. 8.1

[FV94] C. Fahrner and G. Vossen. A survey of database design transformations based on the entity-relationship model. Technical Report 14/94-1, Institut für Wirtschaftsinformatik, Universität Münster, 1994. 4.1.1.4

[FV95] C. Fahrner and G. Vossen. A Survey of Database Design Transformations Based on the Entity-Relationship Model. *Data & Knowledge Engineering*, 15:213 – 250, March 1995. 4.1.1.4

[Sch91] O. Schreyer. Ein Konzept zur automatischen Übersetzung eines Herm-Datenbankentwurfs in eine algebraische Spezifikation, 1991. Diplomathesis (in German). 6.1

[Ste95] M. Steeg. CoDO – The Cottbus Conceptual Database Optimizer – and its Extensible Rule Model EC^{2}A. Technical Report I - 2/1995, Cottbus Technical University, Computer Science, July 1995. 3, 5.3.2, 7.1

[Ste96] M. Steeg. The Conceptual Database Design Optimizer CoDO – Concepts, Implementation, Application. In *[Tha96]*, 1996. 3, 4.3, 5.4, 6.3.1, 3