Eldorado : architecture of a functional database management system

Eindhoven University of Technology

Department of Mathematics and Computing Science

Eldorado: Architecture of a Functional Database Management System

by

A.T.M. Aerts K.M. van Hee

Computing Science Note 91/15
Eindhoven, September 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Eldorado: Architecture of a Functional Database Management System

A.T.M. Aerts* and K.M. van Hee†
Department of Mathematics and Computing Science
Eindhoven University of Technology
Eindhoven, The Netherlands

Abstract

Eldorado is a functional database management system. It has a modular structure, the top layer of which is formed by a graphical interface, called ElView, for data modeling and defining the database structure, and a data language interface, called ElDa, for manipulation of data. Although the data language underlying ElDa is functional in nature, capabilities have been added for defining complex objects, so that ElDa not only can act as a functional, but also as a relational or hierarchical database interface. The underlying database system, the Eldorado Toolkit, has been made sufficiently general to handle unstructured data, such as text or program code, and it has served as the basis for a hypertext system.

1 Introduction

Eldorado is a database management system for supporting functional databases [SHI81, BUN79]. It offers facilities for managing data that are structured as (finite) sets or functions (mappings) between sets. On top of these facilities a data language has been defined for creating databases and maintaining and querying them.

1.1 The functional data model

The functional data model is a flexible and mathematically simple model. It has only a few basic, yet very powerfull constructs. The functional data model we will use here [AH89a] is related to the original proposal by David Shipman [SHI81]. In his 1981 paper, Shipman builds his model around the basic concepts of objects or entities ("unique things that exist" in real life) and functions (relations between objects). The essential difference between Shipman's data model and the functional data model used here is, that Shipman's functions are multivalued in the sense that a function, when applied to an object in its domain, always will yield a set of objects. The functions,

*email: wsinatma@win.tue.nl
†email: wsinhee@win.tue.nl

as defined in [AH89a], are always single-valued, i.e., they yield a single object. The reason for this choice is that, in practice, most of the time mono-valued functions are encountered. Furthermore, multivalued functions can always be modeled using single-valued functions: a mono-valued function when applied inversely also yields a set. A third advantage of the use of single-valued functions is the fact that they have nice implementation properties (see below). A further difference is that Shipman also treats data types, such as string and integer types, needed only for representing names and numbers, as object types. In the functional data model used here, the modeling and representational issues are separated. The *conceptual schema* has two components: a structure schema and a representation schema.

When constructing an instance of the functional data model for a particular object system, one classifies the objects in the system into *object types*, according to the properties they have. An object type has a label or name and is, mathematically speaking, a set. Properties of objects are given by relating one object to another one, and can be regarded as ordered pairs of objects. The first element of each pair is the object having the property; the second element is the object giving the details of the property. Because pairs of objects may be related in more than one way, the ordered pairs have to be labeled, in order to distinguish the various relations. Properties with the same label are collected into functions, i.e., sets of ordered pairs which are characterized by the fact that the first element of any pair is unique within the set. Functions are classified into *property types*, i.e., sets of functions with the same label. A *structure schema* (see Appendix A) specifies which object types are included in the data model and what their properties are, i.e., what kind of functional relationships exist between the various object types. In addition one can specify a number of "standard" constraints, restricting property types to contain only total, injective and/or surjective functions and object types to be subtypes of other object types, as well as specify key- and mutual exclusion constraints. Other constraints will be specified at the representational level, using the data language [AH89a].

In the next stage of the modeling process, the representations for the object types are specified in the *representation schema*. The representation of an object type may be structured according to a relational, a network or a hierarchical data model[AH89b], anticipating the implementation of the database system with a relational, network or hierarchical database management system, respectively. In that case we will have to transform the functional structure schema into the appropriate representation schema. In this paper we will choose for a direct, i.e., a *functional representation model*, which will use the same basic ingredients for the representations as for the structure schema: sets and functions.

We will distinguish between *basic* and *derived* representations. In the former case objects are represented directly by associating them with a (possibly complex) value. In the latter case objects are represented by tuples, constructed from the representations of objects from other object types. Every object type may have a basic representation, but only object types for which (total) key constraints have been defined (see Appendix A) may have a derived representation. A typical example of an object type which derives its representation from other object types is an object type modeling a binary [ABR74, NIJ77] or, in general, an n-ary [CHE76] relationship between object types. An n-ary relationship between object types is modeled by an object type and n

properties that connect the object type modeling the relationship to each of the object types participating in the relationship. An object representing an element of, for example, a ternary relationship will typically be identified by specifying the three objects participating in the relationship. The representation of the former object then may be constructed from the representations of the latter three. On the derivation of the representation of an object type one has to impose, of course, the restriction that no cycles occur. An object type cannot be represented, directly or indirectly, in terms of itself.

A *database state* in the functional model then is specified by giving the *state* of every object type — i.e., by giving the representations of the objects present in the system at that point — and the *state* of their properties. This information can be represented in a *state graph*, i.e., a labeled directed graph with the objects as nodes and the properties as labeled, directed edges:

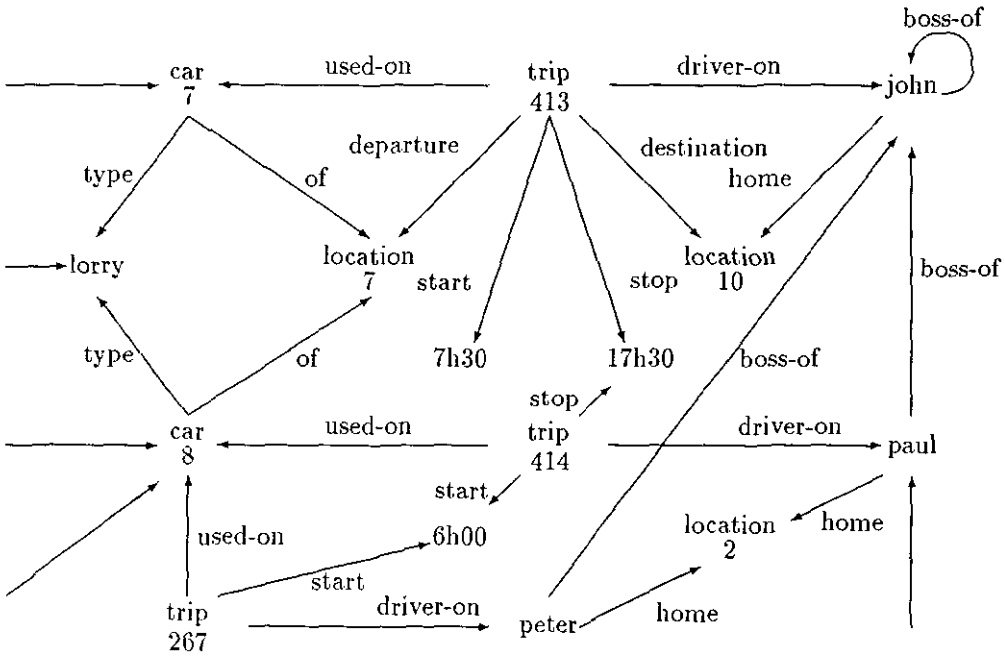

Figure 1: Part of a state graph for a transportation database

A database state has to satisfy the minimal requirements that objects have a valid representation and that properties in a given database state refer to objects, that exist in the same database state. In addition, the standard constraints, specified in the stucture schema, have to be satisfied. The set of database states, that satisfy these requirements, is called the *state space*. We can restrict the state space to become a *restricted state space* by further imposing constraints, using a data language.

1.2 The Eldorado System

The Eldorado System is a database management system for supporting functional databases. It consists of a number of components that together enable one to define and manipulate functional databases, as defined in [AH89a]. In Section 2 we will discuss the components that can be applied to a generic database, be it an ordinary database or a data dictionary (a special kind of database containing meta data). In Section 3 we will discuss the more specialised components.

The Eldorado System was designed to be as general as possible, while at the same time having an acceptable performance. The first objective led to the choice of the functional representation model as the underlying data model. Data are decomposed into the smallest possible units, yielding objects and properties as the elementary data items to be manipulated. With these elementary building blocks more complex structures can be assembled. The structuring of these complex objects can be done using the definition capabilities of ElDa, the data manipulation language. To support the second objective, operations on data have been implemented in the system as much as possible as *collective* operations. Retrieval and update operations are performed setwise and not an-item-at-a-time.

The Eldorado System has been implemented in Modula-2. The code consist of a number of modules which perform functionally related tasks or act on a particular data type. The import relations between the modules result in a layered structure which is such that a module at a certain level imports only from modules at lower levels.

A schematic view of the Eldorado System Architecture is given in Figure 2:

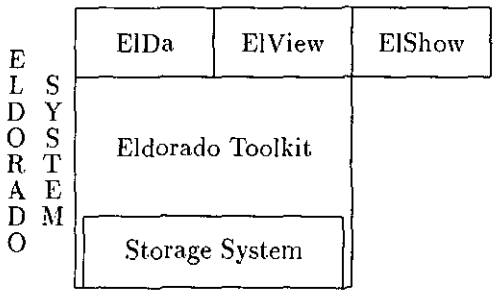

Figure 2: Eldorado System Architecture

A number of features have been added, which make the system more widely usable. The states of the object types have been implemented as sets, on which for efficiency reasons an ordering has been defined based on the ordering of the values from the domain specified in the representation schema for the object type. This ordering is type dependent. However, if objects in one object type have a counterpart in another type with the same domain, the ordering relations of corresponding objects in the two object types will be the same, of course. In addition to a value, an object may have an amount of untyped data associated with it, called an *extension*. This extension may

be retrieved from the database as a by-product of the retrieval of the item from the database. The extension data may be of any kind : text or graphics or even program code. Its use will be determined by the application and is of no concern to the DBMS. This feature has been exploited in the implementation of an hypertext system, called PDB [LEM89], on top of the Eldorado System.

In the next section we will discuss the Eldorado toolkit [LEM87], which comprises the storage system and the toolkit kernel, consisting of procedures for querying and updating objects and properties. The toolkit also provides procedures for manipulating the data dictionary of the system (also a database), as well as procedures for manipulating temporary data structures. The toolkit can be used directly from a user interface, called ElShow, for defining and manipulating a database. In Section 3 we will discuss the data language which has a graphical and a textual component, called ElView and ElDa [HAE89] respectively. In Section 4 we will comment on applications of the Eldorado system and mention current research issues.

2 Eldorado Toolkit Architecture

In the Eldorado Toolkit environment two kinds of data structures are distinguished : temporary and permanent data structures. The permanent datastructures contain the database state and are managed by the storage system. We will refer to the state of an object type as a *database set* and the state of a property type as a *database function.* For manipulation of the database sets and functions temporary data structures have been defined in a layer above the storage system. All data are constructed from the same elements which are called the atoms :

Atoms are the building blocks of all data. They are available in the following types : *integer*, *real* and *string*, which are the types that are also used for the objects in the database, *boolean*, to be used for expressions that evaluate to true or false, *empty*, denoting objects without a value, and *ref*.

A ref indicates an object in the database. Every object is uniquely identified by a ref. In fact, a particular database set can be regarded as a set of refs.

2.1 Storage System

At the storage level we no longer have objects or properties any more. We have items and links between items. Every item is unique, just as the object corresponding to it at a higher level. An item is represented in Eldorado by a unique (internal) identifier, called a *reference*, or *ref* for short. A link, corresponding to a property at a higher level, then can be represented by a triple $< l, ref_1, ref_2 >$ where l is a label, uniquely identifying the property type, and ref_1 and ref_2 are item identifiers, the first one corresponding to the object in the domain of the property type, the second one to the object in the range. An item may also have a *value* and an *extension* associated with it. At the storage level the database state has the structure of a labeled, directed graph, in which the items are the nodes and the links are the edges.

The information in the database is stored in four files: an index file for access by means of the values of the objects, a reference file to store the links among the items, an object file to find values or extensions associated with the items and a location file that holds the locations of items in the reference and object files.

The index is a two level B-tree. The first level tree is a complete index for locating the object type to which an item belongs. With each object type entry a value tree is associated, which is a complete index for the values present in a given database state for that object type. The index is searched by specifying an object type and a value; the position of the corresponding item in the location file (see below) is returned, if present.

The records in the reference file contain a tuple of the form $<id, fr, br>$, where *id* is the ref of the item concerned. *fr* is a list of forward (functional) references : it contains for every property with label *l* which has the object represented by *id* as domain element a tuple $<l$, *rng-el*, *next-dom-el*$>$, where *rng-el* is a pointer to the position in the location file of the object which is the range element of that property, and *next-dom-el* is a pointer to another object which is also mapped by a property with label *l* to the same range object, *next-dom-el*. This structure supports navigation of the structure diagram by means of forward function application : given an object and the property name one can reconstruct the link, if present, and find the range object. It also enables one to find all objects which are mapped to the same range object by a property of a given name. This is useful for reconstructing the complete original for that range object under inverse function application. The list is sorted. The companion of *fr* is *br*, which is a list of backward function references : for every property with label *l* for which the object with ref *id* occurs as a range object a pair $<l$, *dom-el*$>$ is recorded where *dom-el* is the starting element of a sorted list of references to objects that are mapped by properties with name *l* to the object with ref *id*. By going from the object, referred to by *dom-el*, to the object, referred to by *next-dom-el* in the tuple with label *l* in the forward list of *dom-el*, and so on, one now can construct the complete original of the object *id* under *l*. This structure is therefore closely tied to *fr* and supports inverse function application. Note that inverse function application always yields a set of objects.

An item may also have an entry in the object file, where the value and extension are stored. Only objects with a basic representation have values associated with them in the object file. Those with a derived representation do not. Their representation has to be reconstructed by following the forward references correponding the their primary key properties.

The position of items within the files may change as other items are added or removed and unused file space is reclaimed by the garbage collection proces. When we want to use their mutual connections we need to keep track of the items as they move. For this purpose the locations file has been added, that contains the locations of all items in the reference and object files, if present. For every object *id* one has a tuple of the form $<id$, *rloc*, *oloc*$>$, where *rloc* and *oloc* are the locations in the reference and object files, respectively. It will be updated every time a location is changed by the garbage collection or update operations. The data in this storage scheme has intentionally been made redundant, by storing the item identification and the length of reference lists

with every item. This is done for reasons of reliability and efficiency: if part of the stored data is damaged, much of it can be reconstructed by inspecting the undamaged part. If the location file were destroyed, it could be reconstructed by going through the reference and object files. The redundancy further limits the number of disk accesses needed, especially those to the data dictionary.

2.2 Temporary Data Structures

To allow full manipulation of the permanent data, the following temporary data structures have been defined:

Tsets are temporary sets and may be considered as states of a temporary object type. A tset may contain a subset of a database set — in this case it contains refs — or data to be added to it. A tset may be empty or the type of its elements is either integer, real, string or ref.

Tfunctions are temporary functions, that are used for holding updates or retrievals from database functions. Database functions and Tfunctions may be considered as sets of pairs of refs, that link two sets of items.

Tables correspond to the relations of the relational model and are used for the presentation of data. A direct representation of even a part of the state graph would quickly resemble spaghetti.

Tsets, tfunctions and tables are built from atoms or pairs of atoms and can not be used to construct more complex objects recursively. This is done at the level of ElDa.

2.3 Operations

Given the data structures defined above, one needs a collection of operations to transform one structure into another or one type into others. In principle there are many possible choices of operations, but we have concentrated on *collective* operations on database sets and functions. So instead of fetching one item from a particular database set, processing it and then fetching the next one to repeat the processing and so on, all eligible items are extracted from a database set and then processed collectively. Inserting, deleting and searching of items is performed setwise. Updating a database function or set means first building a set of items to be updated and then performing the update operation for the whole set.

Whenever possible, operations will be using *refs* instead of values of items. Only when, after a number of operations, results have to be shown to the user, the values may be determined using a *valuation* operation.

The following operations have been implemented :
Atoms can be compared, using the ordering defined on them. On atoms of type integer and real one can perform arithmetic. Given a ref one can perform a valuation and an extension operation, producing the value and the extension, respectively, that are

associated with the ref. Boolean values can be negated and two booleans can be combined using **and** and **or** into a third boolean.

Temporary sets can be created, united, intersected and subtracted. Elements can be inserted and extracted from them and a tset, containing references, can be valuated, replacing all the refs in the tset by the corresponding values. Aggregate functions on tsets include counting the number of elements, determining in the applicable cases the minimal and maximal element and computing the total and average value and the standard deviation of the elements from the average.

One can determine the minimal and maximal value of the elements of a database set and use this information to perform a range query on the set. Insertion and deletion of objects from a database set is done in two steps. This is due to the fact that an object is represented in the database by a unique identifier, its *reference*, with which a value will be associated in case this object has a basic representation. Objects with a derived representation will just get a reference, as they will derive their value from other objects to which they are linked by a set of key properties. They are also accessed via their key properties. One inserts new objects into a database set by first creating a set of items — one item for each object to be added — and expanding the database set with this set of items. Then, in the next step, the new items may be given values. Deletion of objects from a database set proceeds by removing the values of these objects and isolating the corresponding items. The database set then is reduced by removing these items from the set.

Temporary functions are sets of pairs of references. They can be created in two ways: one can create an empty tfunction and then add pairs of refs to it. Or they can be created by extracting a part of a database function. Tfunctions can be composed to yield another tfunction; one can determine their domain and range as sets of refs, which are subsets of the corresponding (database) sets of items, and apply them to a ref from their domain to yield a reference in their range.

Database functions can be applied to sets of refs from their domain to yield the corresponding subset of their range. They can also be applied inversely. One can add and subtract a tfunction from a database function, in order to insert or delete properties, respectively. Finally, one can restrict a database function to a given subset of its domain. The result is a temporary function.

Tables may be created or recreated, in which case the previous version is removed first. One may add a column to a table by specifying its name and content, by means of a tfunction. The corresponding values can be obtained by applying a valuation operation.

Temporary sets and functions may be duplicated by use of copy operations. The set of operations mentioned above is not minimal in the sense that some composite operations can be constructed in more than one way. Since an application will require more than one database, e.g., a facts base and a dictionary, all operations mentioned above will require, as an extra parameter, the name of the database to which they are to be applied.

2.4 ElShow

ElShow provides a basic interface to the Eldorado Toolkit, which allows a user to apply the operations defined above and certain combinations of these operations to Eldorado databases.

The interface is structured as a tree, the nodes of which are screens, which guide the user to a particular operation. The tree can only be traversed vertically. One proceeds from a parent-node to a child-node or vice versa, except for returning to the root, which is possible from practically every node. The root of the tree is the *Main Menu*-screen, which offers, apart from an exit from the system, six options, each of which leads to an underlying menu. One option is to specify a particular database (new or existing); the other five allow data definition and manipulation of the current database. One may manipulate :

- definitions of object types : inspection, addition or deletion of object types and modification of their name or data type.
- definitions of property types : inspection of name, domain and range, and addition or deletion of property types.
- sets : intersection, union, difference, duplication of sets and inspection of sets of values.
- objects : inspection, insertion and deletion of values and extensions.
- properties : inspection, insertion and deletion of properties.

The first two of these options enable the user to create a database — and its data dictionary — and to maintain its definition. The last three allow for manipulating the database state. Note that the operations are at the level of sets of objects and properties. Adding an object to an object type with a derived representation means specifying the representations of the objects from which the representation of the former object is derived.

Object and property types can be approached in ElShow only one at a time. In case one wants to manipulate several object and property types at the same time, as is quite usual in queries and updates, one has to make use of the data language ElDa. Elshow, on the other hand, enables one to access the database interactively at a low level, which is quite useful in case of testing modules or repairing damage to the data.

3 The data language

On top of the Eldorado system an interface has been built which has two components : a data definition component, called ElView, which is graphical in nature, and a textual component, called ElDa [HAE89], for data manipulation.

3.1 ElView

Data definition capabilities have been provided in ElShow. However, because of the elementary nature of the operations supported by Elshow, the user of ElShow has to possess a certain knowledge of the Eldorado system. In addition, Elshow does not provide a *direct* overview of the definition of a particular database. In order to overcome these inconveniences, a graphical interface, called ElView, has been designed, which provides:

- a userfriendly access to Eldorado for defining a database and maintaining and presenting a database definition.
- a facility for simple data manipulation and querying by means of logical navigation.

Using ElView, a graphical representation of the structure schema (see Appendix A), called the *structure diagram*, can be created or displayed by specifying a new or existing database name, respectively, and then edited. For this purpose three types of actions are supported :

- selection of the appropriate action
- selection of a position or object in the structure diagram
- specification of a name or value

Actions are selected by means of a mouse from a vertical menu bar which is continuously displayed at the left side of an ElView screen. As far as the DDL is concerned this amounts to creating or deleting an object or property type. In case one has chosen to create an object type, for example, one selects a position for the new type in the graphical window, which comprises the major part of an ElView screen. When the object type has been positioned in the structure diagram it has to be named for which purpose a text window has been included at the bottom of the screen. In the text window one is prompted for the name and type of the object type. Upon completion of the specification a box (the graphical primitive indicating an object type) with the name in it will appear in the structure diagram, centered around the chosen position. At this point one may select the next action. Property types are displayed in the structure diagram as named arrows, with the box of the domain type at their base and the range type at their tip (see Figure 3).

Actions on the structure diagram are translated into the corresponding actions on the data dictionary for the specified database. This way the definition of the database is made persistent and available for future use. At the same time the graphical definition of the structure diagram is stored and kept consistent with the contents of the data dictionary, such that it also is available for future use.

Besides data definition, ElView also has some limited data manipulation facilities : insertion and deletion of objects and properties may be performed. In addition simple graphical queries, which involve only function application to specific objects at this

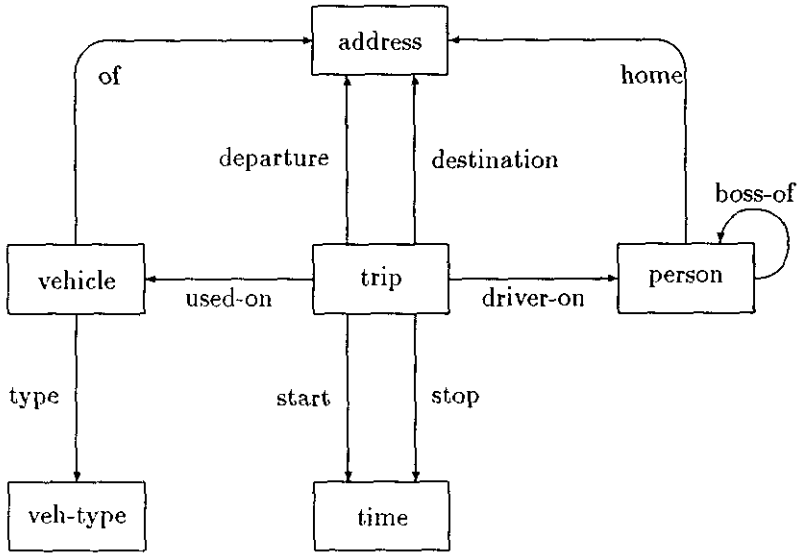

Figure 3: A structure diagram for a transportation database

point, are supported. Using the mouse a path through the structure diagram may be defined that leads from the specified object to the object queried for.

The ElView facilities described above allow one to design and implement a functional database. Since ElView supports data modeling at a semantic level, the structure diagram it produces does not necessarily have to lead to an implementation as a functional database. As a byproduct a tool has been designed and implemented which, given a structure diagram, generates a relational representation, following the algorithm of [AH89b], and an SQL-implementation.

3.2 ElDa

Manipulation of data is supported by ElDa, a language for specifying data manipulation operations. An operation is defined as an expression in ElDa. The expressions are specified, using an interactive editor, and then are interpreted in terms of Eldorado operations and evaluated, using the Eldorado system facilities.

The syntax of ElDa expressions is based on the first order language, proposed in [AH89a] (see Appendix A). The most notable difference between the implementation and the proposal of [AH89a] is the introduction of a range quantor and of the usual operators for doing arithmetic. This part of ElDa allows retrieval, insertion, modification and deletion of (sets of) objects and properties. Queries and updates can be specified by constructing set-expressions that act on the state of the database at hand. For instance:

$$\$[\ p : \text{person} \mid \exists[\ t : \text{trip} \mid p=\text{driver-on}(t) \wedge \text{destination}(t)=\text{home}(p)\ ]]$$

expresses a query (indicated by the $-sign, followed by a set-expression) for all the persons, that were driver on a trip that had as destination the home of the driver. The

result is the set of qualifying persons, as far as registered in the current database state. Updates are specified by ↑ for deletions and ↓ for insertions :

person ↓ { William , Mary }

boss-of ↓ { (William, John),(Mary, John)}

and

driver-on ↑ $[t : trip | John=driver-on(t)]

where William and Mary are added to the set of (known) persons, and John is made boss of both William and Mary. In the third example above all driver-properties are deleted from the database state for trips that had John as a driver. In this example a query has been used to specify the set of objects in the domain of *driver-on* for which the corresponding property has to be deleted.

In addition to being an implementation of [AH89a] ElDa has several elements not present in the proposal. One can temporarily store data by defining free variables and assigning values to them. This is a very useful extension of the language for building sets of objects or properties as a preparation for a complicated update operation, or for keeping intermediate results. It has to be noted that the resulting sets and functions, just as all other variables, are always of the temporary kind. This has the advantage that their values are kept in internal memory and processing is faster than for permanent data, but also the property that these values are lost at the end of an ElDa session.

The data language of [AH89a] distinguishes between two kinds of data: elementary and structured. Elementary data are the atoms of Section 2. They are treated as indivisible, i.e. they cannot be decomposed into smaller parts. Functions and sets are structured data : the data language has operations that enable one to look inside these data elements and determine, e.g., whether an object is member of a particular set. Functions and sets can also be constructed by enumeration of their elements. To these two a third kind has been added in ElDa : the *complex* data type. Complexes can be used to structure data. They can be defined by giving their structure and then be used as expressions in the data language. A complex definition has the following form :

complex < *identifier* >: # < *nucleus* >≪< *funclist* >≫

Here, < *identifier* > is a name, < *nucleus* > is the name of an object type (a permanent data type) and < *funclist* > is a list of (permanent) property types, all of which have < *nucleus* > as their domain or range. In the latter case the function has to be applied inversely. For example, one could cluster the key data for a trip as follows :

complex KeyTrip: # trip ≪ driver-on, start ≫

A complex value of this type could for example be :

≪ driver-on: John, start: 7h30 ≫.

In the example above we have assumed that objects of the type *trip* have a derived representation. In that case, no value for the object in the nucleus has to be (and can be) specified. (As a matter of fact, complexes present a great simplification for manipulating objects with a derived representation.) In case the object type in the nucleus has a basic representation, its elements have to be specified. This is done as follows :

complex Fleet: # veh-type ≪ type ≫

A complex of type *Fleet* has values of the type :

#lorry ≪ type: {car 7, car 8} ≫

Nesting of complex objects is allowed. If for instance *person* would be the nucleus of a complex object, based on the properties *firstname* and *lastname*, then this is expressed as follows:

complex KeyPerson: # person ≪ firstname, lastname ≫

complex KeyTrip: # trip ≪ driver-on * KeyPerson, start ≫.

A complex value of this type then could for example be:

≪ driver-on: ≪ firstname: John, lastname: Wheels ≫, start: 7h30 ≫

We see from this example one application of the complex object : it enables one the cluster data. Complexes can be used in queries and updates and give the possibility to refer to an entire group of related objects and properties in one statement. As such it may serve as a relational interface, where the complex objects have been structured as tuples. However, the tuples allowed in ElDa will, in general, correspond to non-first-normal-form type relations [SCH83] since the values in the tuples may not only be simple, but also tuples or sets of tuples themselves. Complexes in ElDa are restricted to have a tree structure. Traversing the tree one may encounter both forward and inverse function applications. Each time a function is applied inversely, a set of complex structures results.

For communication of definitions and results to the user various display options are supported. In ElDa one can display the values of expressions, which may be simple or complex constants or the result of a query. Furthermore, one has the possibility to display the definition of an object type (its name and data type) or property type, of the database (the conceptual model) or the definitions of all types and variables in a particular session. In case the conceptual model has been defined using ElView, the structure diagram may be displayed in ElDa (but not altered).

Finally, a macro-mechanism is provided to simplify the repetitious entering of combinations of actions on the database. One prepares a text file with ElDa expressions, which can be executed using the **do** command. Do commands can be nested, albeit

not deeper than eight files. One can use this facility for initializing the database environment in which one wants to work by defining a standard set of complex objects and variables, and assigning values to these temporary data objects. Other uses include standard applications such as reporting.

4 Comments and Outlook

Experience with the Eldorado System has shown that building a database system on the basis of sets and functions as basic building blocks is a viable and interesting alternative. The system has proven to be quite general and flexible.

From the user point of view, working with objects and properties tends to become cumbersome after a while, and the possibility of clustering related data into complex objects proves to be a great step towards making the system more userfriendly. The possibility of combining ElDa-expressions in batches in text files furthermore allows a user to tailor the database system to a particular object system and work with smaller or large clusters of data, as is found suitable.

The Eldorado Toolkit subsystem has been used as basis for a hypertext system, called PDB [LEM89]. A hypertext system can be seen as a collection of objects associated with each other by links, which is precisely the view on data underlying the Eldorado System. Objects in a hypertext system have a name and a content (the extension). From a functional database point of view, each hypertext object by itself corresponds to an object type. One thus has very many object types of which never more that one element is present in a particular database state. The B-tree index in this case is effectively reduced to one level and does not provide optimal access to hypertext objects. This index has therefore been replaced by an index based on the Soundex hashing algoritm [KNU73]. Objects and links can be accessed by means of their name. In addition, objects are characterized by keywords, which provide an additional relationship between objects. Typical operations on objects and links supported by the PDB hypertext system include: give the content X of the current object, change the content of the current object to X, give all keywords for the given object, give all objects having a set of keywords containing a given set X, give all range objects of a given object under a relation R, etc. Applications of PDB would include management of text fragments (stored as extensions), which are linked by chapter-section- subsection type relations, and management of program code, on the basis of calling sequences.

The Eldorado System at this point in time is a fairly complete functional database management system, that is useful for the purposes of education and research in the fields of database systems and software engineering. It has been implemented in Modula-2 as a stand-alone system on a PC and is well suited for the purpose of prototyping a database system. It will be interesting, though, to research further various subjects such as the enforcement of the standard constraints, as expressed in the structure schema (Appendix A and [AH89a, AH89b]), in the form of more complex update operations, the support for recursive queries and subtype-supertype hierarchies in the context of ElDa. From a system point of view, the integration ElView with ElDa would be desirable, for the purpose of graphically defining complex objects and derived (temporary) data

types, and queries.

Aknowledgements

It's a pleasure to thank W.J.M. Lemmens and A.M. van den Bent, H.C. Haesen, and L.T.G.M. Thijssen for implemention of the software and many interesting discussions on Eldorado, ElDa, and ElView respectively.

References

[ABR74] **Abrial, J.R.** : *Data Semantics.* in : Data Base Management, J.W. Klimbie and K.L. Koffeman, Eds., North Holland Publ. Co., Amsterdam (1974) 1-60.

[AH89a] **Aerts, A.T.M. and K.M. van Hee** : *A Concise Formal Framework for Data Modeling.* Computing Science Notes 89/12, Eindhoven University of Technology (1989).

[AH89b] **Aerts, A.T.M. and K.M. van Hee** : *Modelleren met een Functioneel Datamodel.* Informatie, 31 (1989) 941-957. (in Dutch.)

[BUN79] **Buneman, P. and R.E. Frankel** : *FQL-A functional query language.* Proc. ACM SIGMOD Conf., Boston, Mass., (1979) 52-58.

[CHE76] **Chen, P.P.** : *The Entity-Relationship Model - Towards a Unified View of Data.* ACM Transactions on Database Systems, 1 (1976) 9-36.

[HAE89] **Haesen, H.C.** : *ELDA, Data Manipulatie Taal.* Computing Science Notes 89/14, Eindhoven University of Technology (1989).

[KNU73] **Knuth, D.E.** : *The Art of Computer Programming, Vol. 3 : Sorting and Searching.*, Addison-Wesley Publ. Co. (1973)

[NIJ77] **Nijssen, G.M.** : *Current Issues in Conceptual Schema Concepts.*, in Nijssen (Ed.), Architecture and Models in Data Base Management Systems, North Holland (1977).

[LEM87] **Lemmens, W.J.M.** : *Eldorado ins and outs. Specification of a database management toolkit according to the functional model.* Computing Science Note 87/11, Eindhoven University of Technology (1987).

[LEM89] **Lemmens, W.J.M.** : *Hypertext als Hulpmiddel.* Elektronica 89/10 (1989) 16-29. (in Dutch.)

[SCH83] **Schek, H.J. and M.H. Scholl** : *The NF^2 Relational Algebra for a Uniform Manipulation of External, Conceptual and Internal Data Structures* in J.W. Schmidt (Ed.), Sprachen für Datenbanken, IFB 72, Springer (1983).

[SHI81] **Shipman, D.W.** : *The Functional Data Model and the Data Language DAPLEX.* ACM Transactions on Database Systems 6, (1981) 140-173.

A Definitions for FDM

Definition 1. *Structure Schema*
A structure schema is a triple $< O, P, C >$ where

O: a finite set of names of object types.

P: a finite set of property types; $P =< F, D, R >$, where

- F is a finite set of names of property types : $O \cap F = \emptyset$.
- D is a function which maps a property type to the object type which is called the *domain type* of the property type; so $D \in F \rightarrow O$.
- R is a function which maps a property type to the object type which is called the *range type* of the property type; so $R \in F \rightarrow O$.

C $= < Q, U, X >$, a triple specifying standard constraints :

- Q : a function which assigns to every property type a number of attributes :
  $Q \in V_{\text{is\_a}} \cup$
  $F \rightarrow \prod(\{<\text{total}, \{\top, \bot\} >, <\text{injective}, \{\top, \bot\} >, <\text{surjective}, \{\top, \bot\} >\})$
  ($\top$ = true, $\bot$ = false; $V_{\text{is\_a}}$ is a set of (is_a-) identifiers, disjoint from F and O).
- U : a function which assigns to an object type $o \in O$ the subsets of $D^{-1}(o)$ of names of property types, that are called the keys of this object type, so $U \in O \rightarrow \mathcal{P}(\mathcal{P}(F))$.
- X : a function which assigns to an object type $o \in O$ the subsets of $D^{-1}(o)$ of names of property types, that have mutually exclusive domains, so $X \in O \rightarrow \mathcal{P}(\mathcal{P}(F))$.

Definition 2. *Functional Representation Schema*
A representation schema for a structure schema $< O, P, C >$ is a triple $< B, G, V >$ where the pair $< B, G >$, called the *domain specification*, is a pair of functions given by :

- $\text{dom}(B) \cap \text{dom}(G) = \emptyset$ and $\text{dom}(B) \cup \text{dom}(G) = O$
- $\text{dom}(B)$ is not empty and contains the object types with a *basic* representation.
- $\text{dom}(G)$ contains the object types with a *derived* representation.
- $B(o)$ is a set (of values) called the *domain* of object type o.
- $G(o) \in U(o)$ is called the *derivation function* and specifies the primary key of object type o; we have $\forall p \in G(o) : Q(p) \cdot \text{total} = \top$.

and the *domain function* V is given by :

- for $o \in \text{dom}(B) : V(o) = B(o)$

- for $o \in \mathrm{dom}(G) : V(o) = \prod \Big(\lambda p \in G(o) : V(R(p))\Big)$

Note that the definition of V is a recursive one. For the definition of the representation schema to be sound, we have to impose on the derivation function G the restriction that it does not introduce any cycles in the identification of an object type, i.e., no object type should represented in terms of its own representation. Objects from types in $\mathrm{dom}(B)$ are represented by (possibly complex) values; those from types in $\mathrm{dom}(G)$ are represented by tuples of (possibly complex) values.

Definition 3. *Conceptual Model*
A conceptual model is a pair consisting of a structure schema and a representation schema.

Definition 4. *Database State*
Given a conceptual model $\ll O, P, C >, < B, G, V \gg$, a *database state* is a function s such that

1. $\mathrm{dom}(s) = O \cup F$.
2. $\forall o \in O : \quad s(o) \subset V(o)$.
3. $\forall p \in F : \quad s(p) \in s(D(p)) \not\rightarrow s(R(p))$.
4. $\forall p \in F : \quad Q(p) \cdot \mathrm{total} = \top \rightarrow s(p)$ is total $\wedge$
   $Q(p) \cdot \mathrm{injective} = \top \rightarrow s(p)$ is injective $\wedge$
   $Q(p) \cdot \mathrm{surjective} = \top \rightarrow s(p)$ is surjective $\wedge$
   $Q(p) \in V_{\mathrm{is\_a}} \rightarrow s(p)$ is total and injective.
5. $\forall p, q \in F : \quad (Q(p) \in V_{\mathrm{is\_a}} \wedge Q(q) \in V_{\mathrm{is\_a}} \wedge Q(p) = Q(q)) \rightarrow$
   $\rightarrow \Big(R(p) = R(q) \wedge \mathrm{rng}\Big(s(p)\Big) \cap \mathrm{rng}\Big(s(q)\Big) = \emptyset\Big)$.
6. $\forall o \in O : \quad \forall Key \in U(o) :$
   $\forall x, y \in \bigcap_{f \in Key} \mathrm{dom}(s(f)) : \Big(\forall f \in Key : f(x) = f(y)\Big) \rightarrow x = y$.
7. $\forall o \in O : \forall f_1, f_2 \in X(o) :$
   $f_1 \neq f_2 \rightarrow \Big(\mathrm{dom}(s(f_1)) \cap \mathrm{dom}(s(f_2)) = \emptyset\Big)$

Definition 5. *First Order Language*
Let $\ll O, P, C >, < B, G, V \gg$ be a conceptual model, and let $\mathbb{D} = \cup\{V(o) \mid o \in O\}$. Its first order language [1] (FOL) L_F then consists of the following elements :

i) *Alphabet*

The alphabet is the union of the following sets of symbols

- constants : $\mathbb{D} \cup O \cup F$

[1] An interpretation function for this language and the data language based on it (see Definition 6) is given in [AH89a]

- variables : $\{X, Y, Z, X_1, Y_1, Z_1, ...\}$
- function symbols : $\{\circ, \cdot, ^{inv}, \text{dom}, \text{rng}, \restriction\}$
- set symbols : $\{\cup, \cap, \setminus, \div\}$
- atom comparison symbols : $\{\leq, =, \geq\}$
- set comparison symbols : $\{\subset, \supset, =\}$
- function comparison symbols : $\{\subset, \supset, =\}$
- atom-set symbols : $\{\in\}$
- logical symbols : $\{\wedge, \vee, \neg, \rightarrow, \leftrightarrow\}$
- quantors : $\{\forall, \exists, \$, \mathcal{N}\}$
- interpunction symbols : $\{[,], \{, \}, (,), —, :, ;, ,\}$

ii) *Terms*

- every $a \in \mathbb{D}$ is an a-term
- every variable is an a-term
- every $o \in O$ is an s-term
- every $f \in F$ is an f-term
- if $a_1, ..., a_n (n \in \{1, 2, ...\})$ are a-terms then $\{a_1, ..., a_n\}$ is an (enumerated) s-term
- if $a_1, ..., a_n, b_1, ..., b_n$ are a-terms then $\{(a_1; b_1), ..., (a_n; b_n)\}$ is an (enumerated) f-term
- a-, s- and f-terms are terms
- if f and g are f-terms, then $f \circ g$ is an f-term
- if f is an f-term and x is an a-term, then $f \cdot x$ is an a-term and $f^{inv}x$ is an s-term
- if f is an f-term, then $\text{dom}(f)$ and $\text{rng}(f)$ are s-terms
- if f is an f-term and s an s-term, then $f \restriction s$ is an f-term
- if f is an f-term and s an s-term, then $f \cdot s$ and $f^{inv}s$ are s-terms
- if s_1 and s_2 are s-terms and θ is a set symbol then $s_1 \theta s_2$ is an s-term
- if X is a variable, $o \in O$ and q a predicate, then $\$[X : o|q]$ is an s-term
- if X is a variable, $o \in O$, $i_1, i_2 \in V(o)$ and $\theta_1, \theta_2 \in \{<, \leq\}$, then $\mathcal{N}[\ X\ :\ o \mid i_1\ \theta_1\ X\ \theta_2\ i_2\ ]$ is an s-term
- there are no other terms

iii) *Predicates*

- if a_1 and a_2 are a-terms and θ is an atom comparison symbol then $a_1 \theta a_2$ is a predicate
- if a is an a-term, s an s-term and θ a atom-set symbol, then $a \theta s$ is a predicate
- if s_1 and s_2 are s-terms and θ a set comparison symbol, then $s_1 \theta s_2$ is a predicate
- if f_1 and f_2 are f-terms and θ a function comparison symbol, then $f_1 \theta f_2$ is a predicate
- if q_1 and q_2 are predicates and θ is a logical symbol, not equal to $\neg$, then $(q_1 \theta q_2)$ is a predicate
- if q is a predicate, then $\neg q$ is a predicate
- if X is a variable, q a predicate and $o \in O$ then $\forall[\ X\ :\ o \mid q\ ]$ and $\exists[\ X\ :\ o \mid q\ ]$ are predicates

- there are no other predicates

Definition 6. *Data language*
Let $\ll O, P, C >, < B, G, V \gg$ be a conceptual model and let L_F be the first order language, with alphabet extended with the symbols $\uparrow$, $\downarrow$ and ;. The data language L_D contains L_F and the following elements

i) *constraints*

Every predicate in L_F, without free variables is a constraint. L_C denotes the sublanguage of L_F containing only constraints.

ii) *queries*

Let $X_1, ..., X_n$ be distinct variables and let $o_1, ..., o_n$ be elements of O and q a predicate with at most $X_1, ..., X_n$ as free variables then

$\$[\ X_1\ :\ o_1, ..., X_n\ :\ o_n \mid q\ ]$

is a query. L_Q is the sublanguage of L_D containing only queries.

iii) *updates*

Let $o \in O$, $f \in F$, and let σ, σ' be s-terms and ϕ, ϕ' be f-terms without free variables, such that σ and ϕ are both enumerated, then:

$o \downarrow \sigma$, $o \uparrow \sigma'$, $f \mid \phi$, $f \uparrow \phi'$

are updates. If u_1 and u_2 are updates then $u_1;u_2$ is also an update. L_U is the sublanguage of L_D containing only update expressions of the form above.

Definition 7. *State Space*
Given a conceptual schema $\ll O, P, C >, < B, G, V \gg$ with data language L_D and a constraint CO $\in$ L_D, then the *State Space* S is defined as :

$S = \{\ s \mid s$ is a database state and satisfies CO $\}$

B Notation

Definition *Generalized Product*
Let P be a set valued function, then the *generalised product* of P is given by :
$\prod\left(P\right) = \{\ p \mid p$ is a function with domain dom(P) and $\forall x \in \text{dom}(P) : p(x) \in P(x)\}$

Definition *Partial Function*
Let A and B be sets, then a function f is a partial function from A to B iff dom$(f) \subset A$ and rgn$(f) \subset B$. This is denoted as : $f : A \nrightarrow B$.

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. .
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.