The R2-algebra : an extension of an algebra for nested relations

Citation for published version (APA):
Houben, G. J. P. M., & Paredaens, J. (1987). *The R2-algebra : an extension of an algebra for nested relations*. (Computing science notes; Vol. 8720). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1987

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

The R^2-Algebra :
An extension of an algebra for nested relations

by

G.J. Houben

J. Paredaens

87/20

december 1987

COMPUTING SCIENCE NOTES

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands
All rights reserved
editor: F.A.J. van Neerven

THE R^2 -ALGEBRA : AN EXTENSION OF AN ALGEBRA FOR NESTED RELATIONS

GEERT-JAN HOUBEN

Eindhoven University of Technology

JAN PAREDAENS

University of Antwerp UIA

november 1987

1. INTRODUCTION

The original formalisms for expressing queries on relational databases come from a mathematical background. One of these formalisms is the relational algebra. Since the introduction of such formalisms however, there has been an evolution in querying relational databases and the need has grown for stronger formalisms. The three main aspects, in which the original formalisms are not strong enough, are :
(1) the presence of the first normal form (1NF);
(2) the absence of aggregation and computation;
(3) the absence of recursion.

Since the introduction of the recommendation that relations should be in first normal form ([C70]), i.e. only atomic data values are allowed, it has been suggested ([AB84], [FT83]) that this widely accepted requirement is in fact too restrictive. Recently a number of proposals ([SS86], [V87]) for nested relations (relations with relational data values) have been made.
Furthermore, it seems that allowing the usage of aggregation, computation and recursion in queries is another step towards a formalism for expressing queries, that is strong enough to serve much more of the user's needs that the original formalisms, like the relational algebra, can do.
Our aim is to define a formalism for expressing queries on a relational database, that comes from a mathematical background, but that adds to the usual features of such formalisms the three features, that we just described.

Since we think that for too long these formalisms have been one dimensional, there is one other important aspect that we want to capture with our formalism. When we say that these formalisms are one dimensional, we mean that systems based on these formalisms are only able to handle queries, that are expressed by one dimensional formulas, i.e. strings of symbols.
This idea of communicating in a one dimensional way with computers is now gradually being replaced by the idea of communicating through a graphical interface, that not only represents the output of the computer in a two dimensional graphical way, but that also handles the computer's input ((answers on) questions to the user) in such a way. Since the best way to express nested relations is in a two dimensional way (e.g. trees), our formalism should allow the user to express queries on relations two dimensionally, i.e. by making full use of the screen of the machine on which the relational system is operating.
The formalism that we will present here is constructed having in mind, that the system that uses this formalism should be two dimensionally, but the formalism itself is presented in the usual mathematical,

i.e. one dimensional, way. Since our formalism is a two dimensional version of the relational algebra we call this formalism the R^2-algebra. As notations for the operations of the R^2-algebra we use one dimensional formulas, but these formulas are chosen in such a way that the two dimensional idea behind an operation (i.e. the two dimensional manipulations on the screen) is obvious. Note that this can imply that these formulas are rather complex from a one dimensional point of view.

2. TWO DIMENSIONAL ALGEBRA

The R^2-algebra has a lot in common with other nested algebras. When defining this algebra we want to capture a number of features, that are not (sufficiently) captured in other nested algebras. One of these features, the two dimensional usage, will be discussed in this section, since a number of definitions strongly depend on this notion. Note however, that we can capture the other features, like having non-1NF relations and allowing aggregation, independently of the introduction of the two dimensional usage into our formalism.

The idea of the R^2-algebra is that a relation is represented by a two dimensional figure on the screen.
When a user wants to express a query, he manipulates a number of known relations in such a way that a new relation is constructed, that holds exactly the information that is specified by the query. In our system we imagine that for a number of relations there are representations on the screen and that with the aid of several screen manipulations a representation of a new relation is constructed on the screen.
The operations of the R^2-algebra will be implemented as performing certain screen manipulations. The system will supply the user with menu's from which operations can be chosen that should be executed, i.e. of which the corresponding screen manipulations should be executed.

A relation consists mainly of two parts, its schema and its value. A relation is used to represent some information. The schema of a relation defines the structure of the information, whereas the value of a relation specifies the current instance of the information.
The schema of a relation defines the attributes of a relation. Also it defines which attributes are atomic, i.e. their value is an atomic data value, and which attributes are structured (nested), i.e. their value is the value of a relation.
We imagine a schema of a relation to be represented on the screen by a tree. Such a tree is constructed by having first of all a vertex labeled with (the name of) the schema. For each atomic attribute of the schema there is an edge to a vertex labeled with (the name of) that atomic attribute and for each structured attribute there is an edge to the tree representing that structured attribute, which is a relation schema itself.
We will be able to manipulate relation schemata (i.e. trees) instead of relations, since we require that every schema corresponds with one value. This implies of course that in the system there is the possibility of asking for the (current) value corresponding with a given tree.
Therefore, the idea is that a user starts with a screen with trees, representing relations. Then he executes operations, which he can choose from several menu's, thus obtaining new trees on the screen, until a tree is obtained that represents the required relation, i.e. the relation that is specified by the query.

3. RELATIONS IN THE R^2-ALGEBRA

In order to define what relations in the R^2-algebra are, we will now introduce some (informal) definitions.
A relation is used to store information. The structure of this information, i.e. the structure of the relation, is described by the schema of the relation. As mentioned in the introduction, we are interested in nested (two dimensional) structures. Such a structure can easily be represented by a tree.

In our formalism a schema is the name of the schema, which is an identifier, followed between parentheses by its attribute list (we use $\|$ for the concatenation of elements in a list).
An attribute is either atomic or structured. An atomic attribute is just the name of that attribute. A structured attribute is a schema.
We define that, if x is a schema or an attribute, the name of x is denoted by $N(x)$, whereas, if x is a

schema or a structured attribute, the attribute list of x is denoted by $L(x)$.
If $n(l)$ is a schema, i.e. n is an identifier and l is an attribute list, then $N(n(l)) = n$ and $L(n(l)) = l$.
If a is an atomic attribute, i.e. a is an identifier, then $N(a) = a$. If a is a structured attribute, i.e. $a = n(l)$ with $n(l)$ a schema, then $N(a) = N(n(l))$ and $L(a) = L(n(l))$.

If l is an attribute list, then $SA(l)$ is defined to be the set of attributes in the list and $SN(l)$ is defined to be the set of the names of the attributes in the list.
We require for every schema that the attributes occurring in the schema have distinct names and that those names are distinct from the name of the schema.

Example :

Let s be the schema A(B(C❙ D)❙ E(F(G❙ H(I))❙ J)❙ K). Then :
$N(s)$ = A, $L(s)$ = B(C❙ D)❙ E(F(G❙ H(I))❙ J)❙ K,
$SA(L(s))$ = { B(C❙ D), E(F(G❙ H(I))❙ J), K }, $SN(L(s))$ = { B, E, K }.
Here A, B, C, D, E, F, G, H, I, J and K are identifiers.
□

$SAA(n(l))$ will be the set of all the atomic attributes of schema $n(l)$. The atomic attributes of schema $n(l)$ are those attributes, that are either atomic attributes in l or atomic attributes of structured attributes in l. So, if a is an atomic attribute, then $SAA(a) = \{a\}$; if a is an attribute and l is an attribute list, then $SAA(a \mid l) = SAA(a) \cup SAA(l)$; if $n(l)$ is a schema, then $SAA(n(l)) = SAA(l)$.
$ALL(n(l))$ will be the set of all the attributes of schema $n(l)$, defined by : if $n(l)$ is a schema, then $ALL(n(l)) = \{n(l)\} \cup ALL(l)$; if a is an atomic attribute, then $ALL(a) = \{a\}$; if a is an attribute and l is an attribute list, then $ALL(a \mid l) = ALL(a) \cup ALL(l)$.

Now we will define what tuples, instances and sets of instances are.
Suppose $n(l)$ is a schema, D is a non-empty set of domains and d is a mapping from $SAA(n(l))$ onto D. A tuple over $n(l)$ (w.r.t. d) is a mapping t, with domain $SA(l)$ and
- $t(a) \in d(a)$, if a is an atomic attribute;
- $t(a) \in I(a)$, if a is a structured attribute.

An instance of $n(l)$ (w.r.t. d) is a set of tuples over $n(l)$ (w.r.t. d). The set of all instances of $n(l)$ (w.r.t. d) is called $I(n(l))$ (w.r.t. d).
For the purpose of this paper we suppose that the mapping d is known, so that it is known for every atomic attribute a what the domain of a is. Therefore, in this paper we do not include D and d in the definition of a relation.
A relation r is a pair $(n(l), v)$, where $n(l)$ is a schema and v is an instance of $n(l)$. We will use $S(r)$ to denote the schema and $V(r)$ to denote the value of a relation r: if $r = (n(l), v)$, then $S(r) = n(l)$ and $V(r) = v$.

Example :

Let STUD be the relation (stud, vstud), where stud is the schema
student(name❙ address(street❙ nr❙ city)❙ year❙ exam(subject❙ attempt(date❙ result))),
and vstud is the instance that could be represented as :

student							
name	address			year	exam		
	street	nr	city		subject	attempt	
						date	result
Bob	Square	1	NY	1	math1	010286	4
	Avenue	88	NY			110486	3
						020387	6
					math2	110186	8
Jim	Road	5	LA	2	compilers	120986	8
						201086	7
					algorithms	111186	1
Bill	Square	1	NY	2	algorithms		

□

4. BINARY OPERATORS

We introduce a number of algebraic operators in order to query the relations of the R^2-algebra. Such an operator is a relation-valued function. In this section we define the binary operators union, intersection, difference and join. These operators have intuitively the same meaning as the binary operators from the (flat) relational algebra. Besides describing the intuitive meaning of the operators, we give the formal (one dimensional) definition and the two dimensional idea behind that definition. Note that for reasons of convenience we define that every available relation has a unique name.

UNION

When we have two relations with the same structure, i.e. with the same schema ($n_1(l)$ and $n_2(l)$ resp.) except of course for the names of the schemata ($n_1 \neq n_2$), then we can use the union operator to compute a relation with that same schema ($n_3(l)$), again except for its name ($n_3 \neq n_1$ and $n_3 \neq n_2$), and with a value that is the (set theoretical) union of the values of the two given relations.

Definition :

Let $r_1 = (n_1(l), v_1)$ and $r_2 = (n_2(l), v_2)$ be relations with $n_1 \neq n_2$. Let $n_3 \neq n_1$ and $n_3 \neq n_2$.
Then we define :

$$\mathrm{UNI}[\ n_1(l)\ ;\ n_2(l)\ ;\ n_3\ ]\ (r_1, r_2) = r_3$$

with $r_3 = (n_3(l), v_3)$ a relation with value

$$v_3 = v_1 \cup v_2\ .$$

□

The notation for the union operator is chosen in this way, since the idea is that when a union of two relations is to be computed, the user first of all has to identify those two argument relations. He does so by clicking the representations of the schemata (i.e. trees) of the relations ($n_1(l)$ and $n_2(l)$) on the screen. The system should react by computing the new relation and by drawing its representation on the screen. Since we require that every relation known in the system has a unique name, the system will ask the user to enter a name (n_3) for this new relation.

Of course, the intersection and the difference can be defined analogously.

INTERSECTION

Definition :

Let $r_1 = (n_1(l), v_1)$ and $r_2 = (n_2(l), v_2)$ be relations with $n_1 \neq n_2$. Let $n_3 \neq n_1$ and $n_3 \neq n_2$.
Then we define :

$\mathrm{INT}[\, n_1(l) \,;\, n_2(l) \,;\, n_3 \,]\, (r_1, r_2) = r_3$

with $r_3 = (n_3(l), v_3)$ a relation with value

$v_3 = v_1 \cap v_2$.

□

DIFFERENCE

Definition :

Let $r_1 = (n_1(l), v_1)$ and $r_2 = (n_2(l), v_2)$ be relations with $n_1 \neq n_2$. Let $n_3 \neq n_1$ and $n_3 \neq n_2$.
Then we define :

$\mathrm{DIF}[\, n_1(l) \,;\, n_2(l) \,;\, n_3 \,]\, (r_1, r_2) = r_3$

with $r_3 = (n_3(l), v_3)$ a relation with value

$v_3 = v_1 - v_2$.

□

JOIN

Basicly, the join can be defined analogously. However, we do not require that the lists of the schemata ($n_1(l_1)$ and $n_2(l_2)$) of the two given relations are the same. The list of the schema of the resulting relation will be composed of the lists of the given relations by concatenating the first list and the part of the second list, that does not occur already in the first list. Again a new unique name (n_3) of the schema must be chosen.

Definition :

Let $r_1 = (n_1(l_1), v_1)$ and $r_2 = (n_2(l_2), v_2)$ be relations with $n_1 \neq n_2$. Let $n_3 \neq n_1$ and $n_3 \neq n_2$.
Then we define :

$\mathrm{JOI}[\, n_1(l_1) \,;\, n_2(l_2) \,;\, n_3 \,]\, (r_1, r_2) = r_3$

with $r_3 = (n_3(l_3), v_3)$ a relation with the list of the schema

$l_3 = l_1 \,\|\, (\, l_2 \upharpoonright (SA(l_2) - SA(l_1))\,)$

and with the value

$v_3 = \{\, t \mid t$ tuple over $n_3(l_3)$ and
$\mathbf{E}(\, t_1, t_2 : t_1 \in v_1$ and $t_2 \in v_2 :$
$\mathrm{A}(\, a : a \in SA(l_1) : t(a) = t_1(a)\,)$ and $\mathrm{A}(\, a : a \in SA(l_2) : t(a) = t_2(a)\,)\,)\,\}$.

□

5. SELECTION AND PROJECTION

Now we will define the operators selection and projection in the R^2-algebra. These operators have the same intuitive meaning as in the flat relational algebra, except for the fact that in the R^2-algebra selection and projection can be applied at all levels of the nested structure.

With the unary operators we often have to specify a number of attributes that play some role in the operation. The two dimensional idea is that the specifying of attributes is done by clicking these attributes on the screen in the desired order. In our formal definition we will give a list of attributes, that represent the clicking of the attributes in the order in which they occur in the list. Since every attribute must have a unique name, we can even give a list of attribute names instead of a list of attributes.

Before defining the selection operator we will give some definitions, that are used in most of the definitions of unary operators.

If *lan* is a non-empty list of attribute names, then $LA(lan)$ is the list of attributes obtained by substituting each attribute name by the attribute, that is uniquely determined by that name.
If a is an attribute and al is a list of attributes, then we use the predicate $O(a, al)$ to denote that a occurs in al. We also use $OS(a, al)$ to denote that the structured attribute a has successor attributes that occur in al. So we define :

$O(a, al) \Leftrightarrow a \in SA(al)$;

$OS(a, al) \Leftrightarrow$ E(a' : $a' \in SA(L(a))$: $O(a', al)$ or $OS(a', al)$) .

Sometimes the attributes that we specify in some list al, must correspond with sibling attributes in a given schema $n(l)$, i.e. the corresponding nodes must be sibling nodes in the representation tree. Therefore we define :

$SIB(al, n(l)) \Leftrightarrow$

E(a : $a \in ALL(n(l))$: A(b : b$\in$SA(al) : O(b,L(a)))) .

Another predicate, that is used in most of the definitions, is the predicate $LO(al, n(l), k)$, that denotes whether there are attributes in the list al that occur in the schema $n(l)$ at level k. It is defined as :

$LO(al, n(l), 0) \Leftrightarrow O(n(l), al)$;

$LO(al, n(l), k) \Leftrightarrow$ E(a : $a \in SA(l)$: $LO(al, a, k-1)$), for $k > 0$.

Now we can introduce the selection and the projection.

SELECTION

The idea of the selection is the traditional idea of selecting tuples satisfying a given criterion. This implies that, given an instance, i.e. a set of tuples, we can compute a subset of that instance determined by a function f and an argument list of attributes al. Also we have the possibility to replace the value of some structured attribute by a subset of that value, when the attributes of al occur at a higher level in the given schema.
We can specify this selection by giving the schema $n(l)$ of the argument relation, the function f and the list of attribute names (lan) that uniquely corresponds with the attribute list al. The attributes must be sibling attributes at least at level 1. The function f and the list of attributes al are such that, given a tuple t over at least the attributes in al, $f(al, t)$ is a boolean value.
Note that in practice one could specify the name of the argument relation instead of its entire schema. Here we will specify the entire schema in order to have all necessary information visible.

Definition :

Let $r = (n(l), v)$ be a relation, f a function, *lan* a list of attribute names and n' an identifier.
Suppose :

$LA(lan) = al$ and $SIB(al, n(l))$ and $n' \neq n$.

Then we define :

SEL[$n(l)$; f ; lan ; n'] $(r) = r'$

with $r' = (n'(l), v')$ a relation, where :

if $LO(al, n(l), 1)$, then

$v' = \{ t \mid t \in v$ and $f(al, t) \}$;

if $LO(al, n(l), k)$ and $k > 1$ and $oa \in SA(l)$ and $OS(o, al)$, then

$v' = \{ t' \mid t'$ tuple over $n'(l)$ and

E(t : $t \in v$:

A(a : $a \in SA(l) - \{oa\}$: $t'(a) = t(a)$) and

$t'(oa) = V($ SEL[oa ; f ; lan ; $N(oa)$] $(oa, t(oa))$)) $\}$.

□

How are we going to apply a selection at a real system ? We start by choosing from some menu of operations the selection operation, i.e. by clicking the SEL operator, and by identifying the argument relation, i.e. by clicking its representation ($n(l)$). The system will react by asking to enter a function (f), i.e. the name of some predefined function, and to click the argument attributes (lan) in the order in which they should occur in the function. When we say clicking an attribute, we mean clicking the root (i.e. the name) of the tree representing that attribute. Then the system knows which criterion is specified by f and lan. Such a selection criterion is a function that, given a list of attributes and a tuple over at least these attributes, determines a boolean value. The functions that can be used as a selection criterion are those functions that can be specified in some programming language (e.g. Pascal) and thus can be computed by the system. The system reacts by computing the new relation and, since there is no new name known, it asks to enter that name (n').

Example :

For illustrating the unary operators, we will use the relation STUD from section 3.

Consider the selection :

SEL[student(name | address(street | nr | city) | year | exam(subject | attempt(date | result))) ; f ; year ; first-year-student]

with f(year, t) $\Leftrightarrow$ t(year) = 1.

When we apply this selection on STUD, then we get a relation S1, which has a schema that is the same as the schema of STUD, except for the name, which is first-year-student. The value V(S1) can be represented as follows :

first-year-student							
name	address			year	exam		
	street	nr	city		subject	attempt	
						date	result
Bob	Square	1	NY	1	math1	010286	4
	Avenue	88	NY			110486	3
						020387	6
					math2	110186	8

So this selection produces an instance with the tuples from the value of STUD, that have the value of year equal to 1. This implies that this selection selects the first-year-students.

Consider the selection :

SEL[student(name | address(street | nr | city) | year | exam(subject | attempt(date | result))) ; g ; attempt ; veteran]

with g(attempt, t) $\Leftrightarrow$ $|\, t(\text{attempt}) \,| \geq 3$.

When we apply this selection on STUD, then we get a relation S2, which has the same schema as STUD, except for the name which is veteran. The value V(S2) can be represented as follows :

veteran							
	address				exam		
						attempt	
name	street	nr	city	year	subject	date	result
Bob	Square	1	NY	1	math1	010286	4
	Avenue	88	NY			110486	3
						020387	6
Jim	Road	5	LA	2			
Bill	Square	1	NY	2			

Applying this selection on STUD produces an instance with tuples, that are obtained from tuples of vstud ($V(STUD)$) by taking in the value of exam only those tuples, that have as value of attempt a set with at least 3 elements. So this selection gives information about the students that have made at least 3 attempts at some subject.
□

PROJECTION

The idea of the projection is again the traditional idea from the relational algebra, i.e. we want to compute a relation, which has a schema that is only a part of the schema of the original relation. The main difference with the relational algebra is the possibility of projecting on attributes at all levels of the structure.
We specify a projection by giving the schema ($n(l)$) and a list of names (lan) of attributes from that schema. The list of attributes specified by lan will be interpreted in the following way : for every attribute occurring in the list, all its successors and all its predecessors occur in the new schema; no other attribute occurs in the new schema. So the tree representing the new schema will be the tree of the original schema, where some entire subtrees are cut away. Of course, the root of the tree (the name of the resulting schema) is also different (n').

Definition :

Let $r = (n(l), v)$ be a relation, lan a list of attribute names and n' an identifier.
Suppose :
$LA(lan) = al$ and $n' \neq n$.

Then we define :
PRO[$n(l)$; lan ; n'] $(r) = r'$
with $r' = (s, v')$ a relation, where :

```
    if $LO(al, n(l), 0)$, then
        $s = n'(l)$,
        $v' = v$;

    if $LO(al, n(l), k)$ and $k > 0$, then
        $v' = \{ t \mid t$ tuple over $s$ and
            E( $t' : t' \in v$ :
            A( $a : a \in SA(l)$ and ( $O(a, al)$ or $OS(a, al)$ ) :
            ( $a \notin SAA(n(l)) \Rightarrow$
                $t(a) = V($ PRO[ $a$ ; $lan \restriction (\{a\} \cup SA(L(a)))$ ] ; $N(a)$ ] $(a, t'(a)))$ ) and
```

$(a \in SAA(n(l)) \Rightarrow t(a) = t'(a)))) \}$;

s is the schema defined by :

$s = n'(PL(l \upharpoonright \{ a \mid a \in SA(l) \text{ and } (O(a, al) \text{ or } OS(a, al)) \}))$,

with $PL(x)$ the projection on a list x, defined by :

if $l = a \mid l'$, then

$PL(l) = S(\text{PRO}[\, a \; ; lan \upharpoonright (\{ a \} \cup SA(L(a))) \; ; N(a) \,] \, (a, \varnothing) \mid PL(l'))$,

if $l = a$, then

$PL(l) = S(\text{PRO}[\, a \; ; lan \upharpoonright (\{ a \} \cup SA(L(a))) \; ; N(a) \,] \, (a, \varnothing))$.

□

On the screen we will first click the PRO operator. After identifying the argument relation ($n(l)$), we have to specify which attributes should occur in the new relation. We could of course click all these attributes, but we have chosen to click only the attributes that, according to the interpretation outlined above, imply all these attributes. So we click some nodes (lan) and the system knows that all these nodes, their successors and their predecessors should occur in the new relation. Of course, we finish the operation by entering a name for the new relation (n').

Example :

Consider the projection :

PRO[student(name ∣ address(street ∣ nr ∣ city) ∣ year ∣ exam(subject ∣ attempt(date ∣ result))) ; name ∣ date ; exam-date].

Applying this projection on STUD produces a relation S3 with schema :

exam-date(name ∣ exam(attempt(date))).

The value V(S3) is :

exam-date	
name	exam
	attempt
	date
Bob	010286 110486 020387
	110186
Jim	120986 201086
	111186
Bill	

So this projection produces an instance with for every tuple from vstud only the value of name and a set, with for every value of subject the set of values of date.

Consider the projection :

PRO[student(name ∣ address(street ∣ nr ∣ city) ∣ year ∣ exam(subject ∣ attempt(date ∣ result))) ; student ; student'].

The application of this projection on STUD results in a relation S4, that except for the name of the schema (student') equals STUD. So this selection produces a copy of STUD and assigns to this copy a new name.

□

6. NEST AND UNNEST

As in all formalisms that allow for relations that are not in first normal form, we have two operators that give the possibility to gain a level in the structure of the schema or to loose a level in that structure.

NEST

The idea behind the nest is that we can take a number of attributes from an attribute list and construct a new structured attribute with exactly those attributes in its attribute list.
So, after specifying the argument relation $(n(l))$, we will specify which attributes have to be nested by giving a list of the names of these attributes (lan). These attributes are required to be sibling attributes. The new attribute must get a name (an), that does not conflict with the names already occurring in the schema.

Definition :

Let $r = (n(l), v)$ be a relation, lan a list of attribute names and both an and n' identifiers.
Suppose :

$LA(lan) = al$ and ($(O(n(l), al)$ and not $OS(n(l), al))$ or $SIB(al, n(l))$) and $n' \neq n$ and $an \notin \{N(a) \mid a \in ALL(n(l))\}$.

Then we define :

NES[$n(l)$; lan ; an ; n'] $(r) = r'$

with $r' = (s, v')$ a relation, where :

if $LO(al, n(l), 0)$, then
  $s = n'(n(l))$,
  $v' = \{ t \mid t$ tuple over $n'(n(l))$ and $t(n(l)) = v \}$;

if $LO(al, n(l), 1)$ and $al' = l \restriction (SA(l) - SA(al))$, then
  $s = n'(al' \mid an(al))$,
  $v' = \{ t \mid t$ tuple over s and
    E(t' : $t' \in v$:
    A(a : $a \in SA(al')$: $t(a) = t'(a)$) and
    $t(an(al)) = \{ u \mid u$ tuple over al and
      E(u' : $u' \in v$:
      A(a : $a \in SA(al)$: $u(a) = u'(a)$) and
      A(a : $a \in SA(al')$: $u'(a) = t'(a)$)) $\}$) $\}$;

if $LO(al, n(l), k)$ and $k > 1$ and $a \in SA(l)$ and $OS(a, al)$, then
  $v' = \{ t \mid t$ tuple over s and
    E(t' : $t' \in v$:
    A(a' : $a' \in SA(l) - \{a\}$: $t(a') = t'(a')$) and
    $t(a) = V($ NES[a ; lan ; an ; $N(a)$]$(a, t'(a))$)) $\}$,

  with the schema s defined by :
    if $l = a \mid l'$, then
      $s = n'(S($ NES[a ; lan ; an ; $N(a)$]$(a, \varnothing)$ $) \mid l')$,
    if $l = l' \mid a$, then
      $s = n'(l' \mid S($ NES[a ; lan ; an ; $N(a)$]$(a, \varnothing)$ $))$,
    if $l = l_0 \mid a \mid l_1$, then
      $s = n'(l_0 \mid S($ NES[a ; lan ; an ; $N(a)$]$(a, \varnothing)$ $) \mid l_1)$,
    if $l = a$, then
      $s = n'(S($ NES[a ; lan ; an ; $N(a)$]$(a, \varnothing)$ $))$.

□

We will start specifying such a nest on the screen by first clicking the nest operator (NES) and subsequently the argument relation $(n(l))$ and the nodes that represent the attributes that are to be nested (lan). The system should of course compute the new relation and draw its tree on the screen while asking to enter a name for the newly constructed node (i.e. the new attribute) (an) and a name for the root (i.e. for the schema) (n'). Note that if the nest is a nest at level 0, then only the new name for the root is relevant.

Example :

Consider the following nest :
NES[student(name | address(street | nr | city) | year | exam(subject | attempt(date | result))) ; street | nr ; str-nr ; student'].
When we apply this operator on STUD, we obtain a relation S5 with schema :
student'(name | address(str-nr(street | nr) | city) | year | exam(subject | attempt(date | result))).
The value V(S5) can be represented by :

student'							
name	address			year	exam		
	str-nr		city		subject	attempt	
	street	nr				date	result
Bob	Square	1	NY	1	math1	010286	4
	Avenue	88				110486	3
						020387	6
					math2	110186	8
Jim	Road	5	LA	2	compilers	120986	8
						201086	7
					algorithms	111186	1
Bill	Square	1	NY	2	algorithms		

This nest rearranges the information in such a way, that the values of street and nr, that occur with one value of city, are taken together in one set, that is the str-nr value corresponding with that value of city.
□

UNNEST

The idea of the unnest is rather the opposite of that of the nest. A structured attribute is substituted by the attributes from its attribute list.
In order to specify this operator we give the name (u) of a structured attribute, that must be unnested. It is obvious that we cannot allow that the schema itself is unnested.
Of course, this operation also requires that we start with identifying the argument relation $(n(l))$ and that we end with specifying a new name (n') for the result relation.

Definition :

Let $r = (n(l), v)$ be a relation, u the name of an attribute and n' an identifier.
Suppose :
$u \neq n$ and $LA(u) = a$ and $a \in ALL(n(l)) - SAA(n(l))$ and $n' \neq n$.

Then we can define :

$\text{UNN}[\, n(l)\, ;\, u\, ;\, n'\,]\,(r) = r'$

with $r' = (s, v')$ a relation, where s and v' satisfy :

```
if $LO(a, n(l), 1)$, then
    $v' = \{ t \mid t$ tuple over $s$ and
        $\text{E}( t' : t' \in v :$
        $\text{A}( a' : a' \in SA(l) - \{ a \} : t(a') = t'(a') )$ and
        $\text{E}( w : w \in t'(a) : \text{A}( a' : a' \in SA(L(a)) : t(a') = w(a') ) ) ) \}$,
    where $s$ is the schema defined by :
        if $l = a \mid l'$, then $s = n'(L(a) \mid l')$,
        if $l = l' \mid a$, then $s = n'(l' \mid L(a))$,
        if $l = l_0 \mid a \mid l_1$, then $s = n'(l_0 \mid L(a) \mid l_1)$,
        if $l = a$, then $s = n'(L(a))$;

if $LO(a, n(l), k)$ and $k > 1$ and $b \in SA(l)$ and $OS(b, a)$, then
    $v' = \{ t \mid t$ tuple over $s$ and
        $\text{E}( t' : t' \in v :$
        $\text{A}( a' : a' \in SA(l) - \{ b \} : t(a') = t'(a') )$ and
        $t(b) = V( \text{UNN}[\, b\, ;\, u\, ;\, N(b)\, ](b, t'(b)) ) ) \}$,
    where $s$ is the schema defined by :
        if $l = b \mid l'$, then $s = n'(S( \text{UNN}[\, b\, ;\, u\, ;\, N(b)\, ](b, \varnothing) ) \mid l')$,
        if $l = l' \mid b$, then $s = n'(l' \mid S( \text{UNN}[\, b\, ;\, u\, ;\, N(b)\, ](b, \varnothing) ))$,
        if $l = l_0 \mid b \mid l_1$, then
            $s = n'(l_0 \mid S( \text{UNN}[\, b\, ;\, u\, ;\, N(b)\, ](b, \varnothing) ) \mid l_1)$,
        if $l = b$, then $s = n'(S( \text{UNN}[\, b\, ;\, u\, ;\, N(b)\, ](b, \varnothing) ))$.
```

□

The usage of the unnest on the screen is rather straightforward. First we click the UNN operator, then the argument relation $(n(l))$ and subsequently the node that represents the attribute that is to be unnested (u) and when the system has computed the new relation it asks to enter the name of that relation (n').

Example :

Consider the unnest :

UNN[student(name | address(street | nr | city) | year | exam(subject | attempt(date | result))) ; attempt ; student'].

Applying this unnest on STUD produces a relation S6 with schema :

student'(name | address(street | nr | city) | year | exam(subject | date | result)).

The value V(S6) of this relation is :

studs							
name	address			year	exam		
	street	nr	city		subject	date	result
Bob	Square	1	NY	1	math1	010286	4
	Avenue	88	NY		math1	110486	3
					math1	020387	6
					math2	110186	8
Jim	Road	5	LA	2	compilers	120986	8
					compilers	201086	7
					algorithms	111186	1
Bill	Square	1	NY	2			

This unnest rearranges the information in such a way that for each exam (i.e. each (date, result)-tuple) the value of subject is stated explicitly. Note how the value "algorithms" disappears in the last tuple, since no exams in algorithms are known.
□

7. AGGREGATION AND COMPUTATION

Now we will introduce two operations, that give the possibility to compute new values based on sets of tuples or on single tuples.

AGGREGATION

What is the idea of the aggregation ? We start with the value of a structured attribute, i.e. a set of tuples. Given some attribute list *al* (specified by a list of attribute names (*lan*)) we compute from that set of tuples a multiset of tuples, by taking for each tuple from the original set of tuples the restriction of the tuple on the attributes of *al*. Then we compute for this multiset an atomic value with the aid of some function f. This new value is stored in a new atomic attribute. We require that the attributes in *al* are atomic attributes and that f is a function, that, given a multiset of tuples over *al*, produces an atomic value.

We can specify an aggregation by giving the argument relation $n(l)$, the function f and the list of attribute names *lan*. We require that the attributes specified by *lan* correspond with sibling nodes at least at level 2. The new attribute, that contains the result value of the aggregation, corresponds with a node, that is a sibling of the parent of the nodes that correspond with the attributes over which is aggregated. The name (*an*) of this attribute must not conflict with the names already occurring in the schema.
Of course we also specify the name of the new schema (n').

Definition :

Let $r = (n(l), v)$ be a relation, f a function, *lan* a list of attribute names, and both *an* and n' identifiers.
Suppose :

$LA(lan) = al$ and $SIB(al, n(l))$ and not $LO(al, n(l), 0)$ and not $LO(al, n(l), 1)$ and $an \notin \{N(a) \mid a \in ALL(n(l))\}$ and $SA(al) \subseteq SAA(n(l))$ and $n' \neq n$.

Then we can define :

AGG[$n(l)$; f ; lan ; an ; n'] $(r) = r'$

with $r' = (s, v')$ a relation, where :

```
    if $LO(al, n(l), 2)$ and $a \in SA(l)$ and $OS(a, al)$, then
        $s = n'(l \mid an)$,
        $v' = \{ t \mid t$ tuple over $s$ and
            E( $t' : t' \in v$ :
            A( $a' : a' \in SA(l) : t(a') = t'(a')$ ) and
            $t(an) = f([ w \upharpoonright SA(al) \mid w \in t'(a) ])$ ) };
        N.B. [ and ] enclose a multiset.

    if $LO(al, n(l), k)$ and $k > 2$ and $a \in SA(l)$ and $OS(a, al)$, then
        $v' = \{ t \mid t$ tuple over $s$ and
            E( $t' : t' \in v$ :
            A( $a' : a' \in SA(l) - \{ a \} : t(a') = t'(a')$ ) and
            $t(a) = V($ AGG[ $a$ ; $f$ ; $lan$ ; $an$ ; $N(a)$ ] $(a, t'(a))))\}$,
        where $s$ is the schema defined by the following :
            if $l = a \mid l'$, then
                $s = n'(S($ AGG[ $a$ ; $f$ ; $lan$ ; $an$ ; $N(a)$ ]$(a, \varnothing)) \mid l')$,
            if $l = l' \mid a$, then
                $s = n'(l' \mid S($ AGG[ $a$ ; $f$ ; $lan$ ; $an$ ; $N(a)$ ]$(a, \varnothing)))$,
            if $l = l_0 \mid a \mid l_1$, then
                $s = n'(l_0 \mid S($ AGG[ $a$ ; $f$ ; $lan$ ; $an$ ; $N(a)$ ]$(a, \varnothing)) \mid l_1)$,
            if $l = a$, then
                $s = n'(S($ AGG[ $a$ ; $f$ ; $lan$ ; $an$ ; $N(a)$ ]$(a, \varnothing)))$.
```
$\square$

The usage of this operator on the screen is similar to that of the selection. After clicking the AGG operator and the schema of the argument relation ($n(l)$), the name of the predefined function (f) is entered and the nodes, that represent the attributes that should be aggregated, are clicked (lan). Subsequently, the names of both the new node (an) and the root (n') must be entered.
The aggregation function is the function f that, given a multiset of tuples (computed with the aid of attribute list $LA(lan)$), determines an atomic value. The functions that can be used as such an aggregation function f are those functions that can be specified in some programming language and are thus computable for the system.

Example :

Consider the aggregation :
AGG[student(name | address(street | nr | city) | year | exam(subject | attempt(date | result))) ; h ; result ; sum ; studsum],
with h the function that assigns to a multiset of integers the sum of the elements.
The application on STUD implies a relation S7 with schema :
studsum(name | address(street | nr | city) | year | exam(subject | attempt(date | result) | sum)).
The value V(S7) is :

studsum								
	address				exam			
name	street	nr	city	year	subject	attempt		sum
						date	result	
Bob	Square	1	NY	1	math1	010286	4	13
	Avenue	88	NY			110486	3	
						020387	6	
					math2	110186	8	8
Jim	Road	5	LA	2	compilers	120986	8	15
						201086	7	
					algorithms	111186	1	1
Bill	Square	1	NY	2	algorithms			0

So this aggregation is used to compute for each student the sum of the results per subject.
□

COMPUTATION

Whereas with the aggregation an atomic value is computed for a set of tuples, with the computation atomic values are computed for separate tuples.
Given a tuple t and a list of attribute names lan that specifies an attribute list al, we use a function f to compute for the restriction of t on the attributes of al an atomic value. This new value is stored in a new additional attribute, which is a sibling attribute of the attributes over which we compute. We require that the attributes in the attribute list al are atomic and that f is a function that, given an attribute list al and a tuple over al, determines an atomic value.
We specify a computation in almost exactly the same way as we specify an aggregation. Since the new attribute becomes a sibling attribute of the attributes over which is computed, we can allow here that they are at level 1.

Definition :

Let $r = (n(l), v)$ be a relation, f a function, lan a list of attribute names and both an and n' identifiers.
Suppose :
$LA(lan) = al$ and $SIB(al, n(l))$ and not $LO(al, n(l), 0)$ and $an \notin \{N(a) \mid a \in ALL(n(l))\}$ and $SA(al) \subseteq SAA(n(l))$ and $n' \neq n$.

Then we can define :
COM[$n(l)$; f ; lan ; an ; n'] $(r) = r'$
with $r' = (s, v')$ a relation, where :

```
if $LO(al, n(l), 1)$, then
    $s = n'(l \mid an)$,
    $v' = \{ t \mid t$ tuple over $s$ and
        E( $t' : t' \in v$ :
        A( $a : a \in SA(l) : t(a) = t'(a)$ ) and $t(an) = f(al, t')$ ) $\}$;

if $LO(al, n(l), k)$ and $k > 1$ and $a \in SA(l)$ and $OS(a, al)$, then
```

$v' = \{\, t \mid t$ tuple over s and
$\quad$ $\mathrm{E}(\, t' : t' \in v :$
$\quad$ $\mathrm{A}(\, a' : a' \in SA(l) - \{a\} : t(a') = t'(a')\,)$ and
$\quad$ $t(a) = V(\, \mathrm{COM}[\, a \,; f \,;\, lan \,;\, an \,; N(a)\,]\,(a, t'(a))\,)\,)\,\}$,
where s is the schema defined by the following :
$\quad$ if $l = a \mid l'$, then
$\quad\quad$ $s = n'(S(\, \mathrm{COM}[\, a \,; f \,;\, lan \,;\, an \,; N(a)\,]\,(a, \varnothing)\,) \mid l')$,
$\quad$ if $l = l' \mid a$, then
$\quad\quad$ $s = n'(l' \mid S(\, \mathrm{COM}[\, a \,; f \,;\, lan \,;\, an \,; N(a)\,]\,(a, \varnothing)\,))$,
$\quad$ if $l = l_0 \mid a \mid l_1$, then
$\quad\quad$ $s = n'(l_0 \mid S(\, \mathrm{COM}[\, a \,; f \,;\, lan \,;\, an \,; N(a)\,]\,(a, \varnothing)\,) \mid l_1)$,
$\quad$ if $l = a$, then
$\quad\quad$ $s = n'(S(\, \mathrm{COM}[\, a \,; f \,;\, lan \,;\, an \,; N(a)\,](a, \varnothing)\,))$.

□

The usage on the screen will be very similar to that of the aggregation.

Example :

Consider the computation :
COM[student(name | address(street | nr | city) | year | exam(subject | attempt(date | result))) ; i ; year ; yearpar ; student'],
with i(year, t) = t(year) (mod 2).
When we apply this computation on STUD we obtain a relation with schema :
student'(name | address(street | nr | city) | year | exam(subject | attempt(date | result)) | yearpar).
The value of this relation V(S8) is :

studs								
name	address			year	exam			yearpar
	street	nr	city		subject	attempt		
						date	result	
Bob	Square	1	NY	1	math1	010286	4	1
	Avenue	88	NY			110486	3	
						020387	6	
					math2	110186	8	
Jim	Road	5	LA	2	compilers	120986	8	0
						201086	7	
					algorithms	111186	1	
Bill	Square	1	NY	2	algorithms			0

So this computation computes for each tuple a value, i.e. the parity of the value of year, and adds it to the tuple.
□

We have now introduced the so called generating operations, i.e. operations that manipulate the information in such a way that new information is produced. In a real system we also need some non-generating operations, like the renaming and reordering of attributes. These operations must give the possibility to manipulate relations, i.e. trees, in such a way that we are able to use the generating operations according to their definitions. In this paper we will not introduce these operations.

8. COMPOSITION OF OPERATIONS

Now we will turn to the possibility of composing the operations introduced so far into new operations. For reasons of convenience we will call the operations introduced so far basic operations. So we are now going to define how operations, which are in fact compositions of (basic) operations, can be defined in the system and how they can be used.
Whereas a basic operation determines a relation, given a relation or a pair of relations, a (self-defined) operation determines a set of relations, given a set of relations.
A user can only use an operation when the system knows that operation. This means that the user must define the operation in the system before being able to use it. An operation can be defined by storing an operation definition in the system. An operation definition defines what the resulting set of relations is, in case the operation is applied to a set of relations.

The formalism that we use for operations is very similar to the one we use for basic operations.
In our formalism an operation is the name of the operation (an identifier) followed by the [symbol, a list of relation schemata, the ; symbol, a list of relation names (i.e. a list of identifiers) and the] symbol.
An operation definition is an operation followed by the := symbol and an operation body. An operation body is the |[symbol followed by a list of (basic) operations, the ; symbol, a list of relation names (i.e. a list of identifiers) and the]| symbol.

For every operation there is exactly one operation definition. This means that for every operation there is exactly one operation definition with that operation on the left hand side of the := symbol. Furthermore every operation must have a unique name.
In a real system a new operation is available iff an operation definition for that operation is stored, so the system is able to compute what the result of the application of this operation is.
The list of relation names in an operation must be equal to that in its operation definition.

As already mentioned, an operation is applied to a set of relations. In the operation the list of relation schemata denotes on which relations the operation is applied.
The list of relation names in an operation determines which relations are in the resulting set of relations. This specification of the resulting set is given in order to be able to neglect relations, that have been computed as an intermediate result, but that are not important for the user.

Example :

The following is an operation definition :
    ps[N(A | B(C | D)) ; N | N"] :=
    |[PRO[N(A | B(C | D)) ; B ; N'] | SEL[N'(B(C | D)) ; f ; B ; N"] ; N | N"]|.
As we will see, after our definition of the application of an operation, this operation definition implies that the operation ps[N(A | B(C | D)) ; N | N"] is available and that it can be applied to a set containing a relation with schema N(A | B(C | D)).
□

Now we will define what the result of applying an operation is. This means that we define what the resulting set of relations is, when applying the operation to a given set of relations.

Definition :

Suppose *o* is an operation and *o* := *b* is the corresponding operation definition. Let *lrs* be the list of relation schemata in *o*. Here we suppose that in any set of relations every relation has a unique name. Let *s* be a set of relations.

We define that *o* applied to *s* equals the union of the set of those relations from *s* for which the schemata are not specified in the list *lrs* and the set of relations obtained by applying operation body *b* to the set of relations from s for which the schemata occur in lrs :

$o(s) = (s - x) \cup b(x)$,
with $x = \{ r \mid r \in s$ and $S(r) \in SS(lrs) \}$ and
$SS(lrs)$ is the set of schemata occurring in the list lrs.

So we now will define what $b(x)$ is.
Suppose $b = |[\ lbo\ ;\ lrn\ ]|$, with lbo a list of (basic) operations and lrn a list of relation names.
It is defined that $b(x)$ equals $|[\ lbo\ ;\ lrn\ ]|\ (x)$, which is defined by :
if bo is a (basic) operation and lbo' is a list of (basic) operations, then

$|[\ bo \mid lbo\ ;\ lrn\ ]|\ (x) = |[\ lbo'\ ;\ lrn\ ]|\ (bo(x))$,
$|[\ bo\ ;\ lrn\ ]|\ (x) = |[\ ;\ lrn\ ]|\ (bo(x))$,
$|[\ ;\ lrn\ ]|\ (x) = \{ r \mid r \in x$ and $N(S(r)) \in SRN(lrn) \}$,
with $SRN(lrn)$ the set of relation names in the list lrn.

We must define what the application of a basic operation bo on a set of relations x is. Of course, we define this as adding to x the relation obtained by applying bo to the relation with the schema that is specified in bo.
So, we define for basic operation bo that

$bo(x) = x \cup \{ bo(r) \mid r \in x$ and $S(r) = S(bo) \} \cup$
$\{ bo(r, r') \mid r \in x$ and $r' \in x$ and $(S(r), S(r')) = S(bo) \}$,

where $S(bo)$ is either the schema of the relation on which bo is applied if bo is a unary operation, or the pair of schemata of the two relations on which bo is applied if it is a binary operation.
Of course, we require that for every (basic) operation in the list of (basic) operations, it can be applied to some relation or set of relations in s.
$\square$

So applying an operation on a set of relations s implies taking the set of relations x specified by the list of schemata. Then the (basic) operations specified by the operation definition are applied to x, thus getting new relations, which are added to x. After all (basic) operations have been applied all those relations are removed from x, for which their name is not specified in the list of relation names. The set thus obtained is added to $s - x$.

Example :

Consider the operation definition

ps[N(A | B(C | D)) ; N | N"] :=
|[PRO[N(A | B(C | D)) ; B ; N'] | SEL[N'(B(C | D)) ; f ; B ; N"] ; N | N"]|.

If r is a relation with schema K(L | M(N | O)), then

ps[K(L | M(N | O)) ; K | P] $(\{r\}) = \{r, r'\}$,

with r' the relation with name P obtained from r by applying the above projection and selection.
$\square$

So operations can be used to store in the system sequences of basic operations that the user wants the system to be able to apply autonomously.
The set of relations on which an operation is applied represents the screen with representations of relations at the start of the operation, i.e. at the moment of specifying on the screen which operation is to be applied. The resulting set of relations represents the screen after the application. So the resulting set specifies the answer on the query, that the user has specified by choosing an operation from the menu of possible operations.

Note that we can imagine that in practice an operation definition is stored by telling the system to memorize a (manually) specified sequence of (basic) operations, in such a way that afterwards the system is able to execute this sequence of operations itself.

9. CONCLUSIONS

In this paper we introduce the R^2-algebra, which in a number of aspects is much stronger than the existing formalisms for expressing queries.
The first aspect, in which the R^2-algebra is stronger, is the possibility of having nested relations. This is convenient, since many database applications can be managed much easier, when nested structures can be used. Furthermore, the selection is stronger. Not only because we can select at higher levels, but also because the criterion for the selection is not as basic as in many formalisms, since it can be programmed according to the user's needs. We also define aggregation and computation, that help to create new values, based on sets of tuples or on single tuples. Again the user is able to program the function that computes the new values.
The user is given the possibility to define new operations as compositions of operations already defined. In this way a user can easily program his queries. Also this should help him to express recursive queries. At the moment we study how recursion can be expressed in the R^2-algebra.
All of this implies that our algebra is of course much stronger than the relational algebra, but also that it is stronger than e.g. the NF^2-algebra of [SS86]. We can show that, although we define the binary operators only at the first level, we can express the binary operations at higher levels in our algebra. The expressive power of the R^2-algebra is another subject of current study.
In the definition of this algebra we also capture the notion that the formalism should be two dimensional. This implies that a system, operating according to such a formalism, is able to manage queries in a two dimensional way. So the system and the user communicate through a graphical interface. Although this implies that our definitions sometimes seem complicated from a mathematical point of view, the system, that we specify with these definitions, gives the user the possibility to express queries, i.e. to manipulate data, in an intuitively much easier way. So not only more database applications can be managed by such a system, but they also can be handled in a more friendly way.

10. REFERENCES

[BA84] S. Abiteboul, N. Bidoit, "Non First Normal Form Relations to Represent Hierarchically Organized Data", Proc. Third ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1984, 191-200.

[C70] E.F. Codd, "A Relational Model for Large Shared Data Banks", Comm. ACM, Vol. 13, No. 6, 1970, 377-387.

[FT83] P.C. Fischer, S.J. Thomas, "Operators for Non-First-Normal Form Relations", Proc. IEEE Computer Software and Applications Conference, 1983, 464-475.

[SS86] H.J. Schek, M.H. Scholl, "The Relational Model with Relation-Valued Attributes", Information Systems, Vol. 11, No. 2, 1986, 137-147.

[V87] D. Van Gucht, "On the Expressive Power of the Extended Relational Algebra for the Unnormalized Relational Model", Proc. Sixth ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1987, 302-312.

86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arum Kumar	Compositional semantics for real-time distributed computing (Inf. & Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responders commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of a sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J. Paredaens	A formal approach to distributed information systems
87/04	T. Verhoeff	Delay-insensitive codes - An overview
87/05	R. Kuiper	Enforcing non-determinism via linear time temporal logic specification

87/06	R. Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch)
87/07	R. Koymans	Specifying message passing and real-time systems with real-time temporal logic
87/08	H.M.J.L. Schols	The maximum number of states after projection
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics
87/10	T. Verhoeff	Three families of maximally nondeterministic automata
87/11	P. Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model
87/12	K.M. van Hee A. Lapinski	OR and AI approaches to decision support systems
87/13	J. van der Woude	Playing with patterns, searching for strings
87/14	J. Hooman	A compositional proof system for an occam-like real-time language
87/15	G. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.J. Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples