Combining the functional and the relational model

Combining the functional and the relational model

by

A.T.M. Aerts P.M.E. de Bra K.M. van Hee

90/9

October, 1990

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 or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands

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

Combining the Functional and the Relational Model

A. T. M. Aerts, P. M. E. De Bra and K. M. van Hee
Eindhoven University of Technology

August 31, 1990

Abstract

This paper combines the expressive power of the Functional and the Relational Model. While the Functional Data Model is a powerful and intuitive tool for translating a real-world situation into a data model, the Relational Model has been studied much more extensively, resulting in a thorough knowledge of the properties of constraints, and the expressive power of query and update languages.

Given the existence of a Relational Representation Scheme for every Functional Structure Scheme we can combine a data language for the relational model with that of the Functional Model to obtain a richer data language for the combined Functional and Relational Data Model.

In this paper we present a first order language which operates on the combined Functional and Relational scheme. In this (formal) language one can easily formulate constraints, queries and updates, mixing functional and relational constructs.

1 Introduction

The Relational Database Model [3, 4, 6] provides a solid theoretical background for reasoning about databases, for formulating queries and updates, and for describing constraints. However, modeling a real-world situation using a relational database is non-trivial at best. One cannot easily and intuitively define relations and constraints that form a useful and efficient representation of the real-world situation. The Functional Data Model does provide such an intuitive tool, and has been proven successful in many database-design projects performed by students. A graphical representation of objects, functions, and several types of constraints enables the designer to generate a visualization of the database scheme that can easily be understood by non-experts.

In order to use the Functional Data Model in real applications, an algorithm has been developed to generate a relational database scheme from a functional scheme. [2] This algorithm is not entirely automatic: the user sometimes has to make a choice whether or not to generate separate relations, mostly when resolving so called *is_a* relationships. In any case the algorithm produces a relational database scheme in Boyce-Codd Normal Form.

The approach until now has always been to model the real world using the functional model, then convert the functional scheme to a relational scheme, and then define constraints, queries and updates on the relational scheme. This is not a desirable approach, since the functional scheme already represents some types of constraints, which have to be reformulated using the relational scheme.

In this paper we describe a new (mathematical) data language, which operates on both the functional and the relational representation of a database. This not only enables the designer to select his preferred model for formulating a query, update or constraint, but also to define them using both representations in one and the same expression.

Because of the subject, and to limit the size of this paper, we assume that the reader is aware of the basic definitions of both the functional data model [2] and the relational model [4, 6].

2 Formalism

The structure of a functional data model for an object system — the part of the real world that is of interest to us — is specified by giving a structure scheme:

Definition 2.1 *Structure Scheme*
A structure scheme is a 4-tuple $< O, P, C, W >$ where

O: a finite set of names of object types.

P: a triple specifying 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:
$Q \in F \rightarrow V_{\text{is\_a}} \cup$
$\prod(\{<\text{total}, \{\top, \bot\}>, <\text{injective}, \{\top, \bot\}>, <\text{surjective}, \{\top, \bot\}>\})$[1]

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 $X \in O \rightarrow \mathcal{P}(\mathcal{P}(F))$.

W: a set valued function with $\text{dom}(W) = O$, called the object world function. If for $f \in F$ it holds that $Q(f) \in V_{\text{is\_a}}$ then $W(D(f)) \subseteq W(R(f))$ □

The components O, P and C specify the object types and the property types, that are included in the data model, and a number of general constraints these types have to satisfy. W is a function which associates a set of real world objects with (the name of) an object type. These sets do not have to be disjoint. On the contrary, if two object types are related to one another by an is_a relationship, the set of real world objects corresponding to the subtype is required to be a subset of the set of objects for the supertype.

The O and P components of the structure scheme specify a labeled, directed graph, in which the object types are nodes and the property types correspond to labeled, directed edges. Not every graph specified this way is acceptable as a structure scheme. When we select the subgraph $<N, E>$ based on the property types with an is_a-label with edges $E = \{f \in F \mid Q(f) \in V_{is\_a}\}$ and nodes $N = \{o \in O \mid \exists\{f \in E \mid o = D(f) \vee o = R(f)\}\}$, this subgraph has to be free from cycles. Furthermore, if there is no directed path from an object type $o_1 \in N$ to an object type $o_2 \in N$ then o_1 and o_2 correspond to disjoint sets of objects: $W(o_1) \cap W(o_2) = \emptyset$.

The is_a properties are required to be total and injective. Therefore we do not repeat them in U like the other injective properties.

Example 2.1 *Student-Professor database*
Figure 1 shows a small database, concerning students and professors:

The components $<O, P, C, W>$ in this example are:

O = {address, name, date, person, professor, student, amount}

P = $<F, D, R>$, where

F = {lives at, called, born on, p-isa, s-isa, loan, scholarship}

D = {(lives at, person), (called, person), (born on, person), (p-isa, professor), (s-isa, student), (loan, student), (scholarship, student}

[1] This symbol denotes the *Generalized Product*: Let P be a set valued function, then $\prod(P) = \{\ p \mid p \text{ is a function with domain } \text{dom}(P) \text{ and } \forall x \in \text{dom}(P) : p(x) \in P(x)\}$, also, $\top$ = true, $\bot$ = false; $V_{\text{is\_a}}$ is a set of (is_a-)labels, disjoint from F and O.

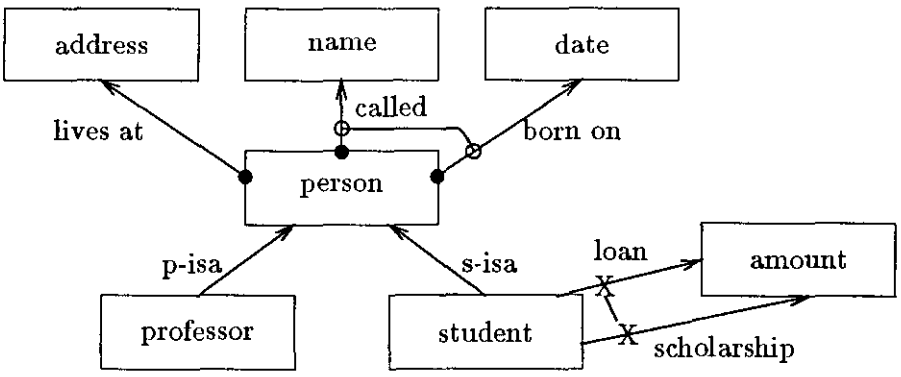

Figure 1: Functional Student-Professor database scheme

R = {(lives at, address), (called, name), (born on, date), (p-isa, person), (s-isa, person), (loan, amount), (scholarship, amount)}

C = $<Q, U, X>$, where

Q = {(lives at, ($<$ total, $\top>$, $<$ injective, $\bot>$, $<$ surjective, $\bot>$)),
(called, ($<$ total, $\top>$, $<$ injective, $\bot>$, $<$ surjective, $\bot>$)),
(born on, ($<$ total, $\top>$, $<$ injective, $\bot>$, $<$ surjective, $\bot>$)),
(loan, ($<$ total, $\bot>$, $<$ injective, $\bot>$, $<$ surjective, $\bot>$)),
(scholarship, ($<$ total, $\bot>$, $<$ injective, $\bot>$, $<$ surjective, $\bot>$)),
(p-isa, $\text{is\_a}_{\text{professor}}$), (s-isa, $\text{is\_a}_{\text{student}}$)}

U = {(person, {called, born on})}

X = {(student, {loan, scholarship})}

W = the function linking the object types in the database to the corresponding set of objects in the real world. This basically is the "meaning" of the database.

From U we see (as shown in Figure 1) that the key properties for a person are his (her) name (property *called*) and date of birth (property *born on*).

X tells us that a student cannot have both a loan and a scholarship at the same time. In some sense, having such properties which exclude each other is an alternative to creating subtypes (using is_a properties) in some simple cases. We could have created subtypes student-with-loan and student-with-scholarship, but even that would not necessarily have meant that a student cannot have both a loan and a scholarship, just as a person can be both a professor and a student at the same time. □

Given a structure scheme, we can describe the states of the object system in terms of a graph called the state graph. The nodes in this graph correspond to real world objects, the edges to the properties that relate pairs of real world objects.

Definition 2.2 *State Graph*
For a structure scheme $<O, P, C, W>$ a *state graph* is a function g such that

1. $\mathrm{dom}(g) = \quad O \cup F.$

2. $\forall o \in O: \quad g(o) \subset W(o)$ and $g(o)$ is finite.

3. $\forall f \in F: \quad g(f) \in g(D(f)) \not\to g(R(f)).$[2]

4. $\forall f \in F: \quad Q(f) \cdot \mathrm{total} = \top \Rightarrow g(f)$ is total $\wedge$
   $Q(f) \cdot \mathrm{injective} = \top \Rightarrow g(f)$ is injective $\wedge$
   $Q(f) \cdot \mathrm{surjective} = \top \Rightarrow g(f)$ is surjective $\wedge$
   $Q(f) \in V_{\mathrm{is\_a}} \Rightarrow g(f)$ is total and injective.

5. $\forall f, h \in F: \Big(Q(f) \in V_{\mathrm{is\_a}} \wedge Q(h) \in V_{\mathrm{is\_a}} \wedge Q(f) = Q(h)\Big)$
   $\Rightarrow \Big(R(f) = R(h) \wedge \mathrm{rng}(g(f)) \cap \mathrm{rng}(g(h)) = \emptyset\Big).$

6. $\forall o \in O: \quad \forall Key \in U(o): \forall x, y \in \bigcap_{h \in Key} \mathrm{dom}(g(h)):$
   $\Big(\forall f \in Key : g(f)(x) = g(f)(y)\Big) \Rightarrow x = y.$

7. $\forall o \in O: \forall Excl \in X(o) : \forall f_1, f_2 \in Excl :$
   $f_1 \neq f_2 \Rightarrow \Big(\mathrm{dom}(g(f_1)) \cap \mathrm{dom}(g(f_2)) = \emptyset\Big)$

□

Basically what this definition says is that a state graph (and consequently the database instance, see Definition 2.5) must satisfy the constraints of the structure scheme.

Using a structure scheme we can describe the structure of the object system. In the next step of designing a relational database we need to specify how we will represent the objects in terms of relations and attributes.

Definition 2.3 *Relational Representation Scheme*
A relational representation scheme for a structure scheme $< O, P, C, W >$ is a 6-tuple $< E, A, V, I, H, T >$ where:

A: a finite set of names of attributes

V: a function with $\mathrm{dom}(V) = A$, which maps every attribute name $a \in A$ to a set of values $V(a)$, called *attribute range*, sometimes also called the *domain* of the attribute.

H: a function, which maps every object type $o \in O$ to a set of attribute names $H(o) \subseteq A$, so $H \in O \to \mathcal{P}(A)$.

I: a function, which maps every object type $o \in O$ to a set of attribute names $I(o)$, which is called the *primary key* of o, so $I \in O \to \mathcal{P}(A)$ and $\forall o \in O : I(o) \subseteq H(o)$.

E: a function, which maps every property type $f \in F$ to an injective function of attribute names to attribute names, such that:

[2] The symbol $\not\to$ is used to denote a *partial function*.

$\mathrm{dom}(E) = F$ and
$\forall f \in F : E(f) \in I(R(f)) \rightarrow H(D(f))$ and
$\forall f \in F : \forall a \in I(R(f)) : V(a) = V\Big(E(f)(a)\Big)$.

T: a function which maps each object type to a function which maps the identifying (primary key-) part of the representation of a real world object to the object itself, so $\mathrm{dom}(T) = O$, and
$\forall o \in O : T(o) \in \prod(V \restriction I(o)) \rightarrow W(o)$.

□

Contrary to what people usually believe a (relational) representation scheme is not just an equivalent scheme in another formalism. It does not represent all information present in the structure scheme, (in particular, it does not repeat the constraints of the structure scheme,) but also, the structure scheme does not uniquely determine the relational representation scheme. The (functional) structure scheme is created in an early design phase, whereas the (relational) representation scheme is developed at a later stage, when getting closer to an implementation, and when filling in more details, like adding additional attributes. So the representation scheme contains new information. To clarify this in our example, we present *a* relational representation scheme for the Student-Professor database:

Example 2.2 *Relational Student-Professor database*
The components $< E, A, V, I, H, T >$ of a possible relational representation scheme are:

$A =$ {street, state, zip, city, address, name, birthdate, SS#, empno, studno, loan, scholarship}
These are the names of the attributes we will use in our relations.

$V =$ {(street, *strings*), (state, *char*[2]), (zip, *numbers*), (city, *strings*), (address, *numbers*), (name, *strings*), (birthdate, *dates*), (SS#, *numbers*), (empno, *numbers*), (studno, *numbers*), (loan, *money*), (scholarship, *money*)}
where *strings*, *char*[2], *numbers*, *dates*, *money* denote the sets of all possible strings, strings of 2 characters, numbers, dates, and amounts of money.

$H =$ {(address, {address, street, state, zip, city}), (person, {SS#, address, name, birthdate}), (professor, {empno, SS#}), (student, {studno, SS#, loan, scholarship})}
H defines headings of the tables. In principle the definition states that a heading must be produced for every object type. However, when no information is lost (this is for the designer to decide) the headings for tables with only one attribute may be omitted. We have chosen here not to define relations for the object types name, date and amount, as the corresponding information can be found in the person or student relations. Note also that we have added attributes (empno and studno) which are not present in the structure scheme. It is common to omit attributes in a structure scheme to trim down the graphical representation. (An alternative is to cut the scheme into logically meaningful subschemes.)

$I =$ {(address, {address}), (person, {SS#}), (professor, {empno}), (student, {studno})}
I defines the primary keys. We have chosen the internal representation of an address, the social security number of a person, the empno of a professor and the studno of a student.

(Again we have omitted the objects with only a single attribute, though the definition requires them.) Note that the objects may have more than one key. The relational representation scheme does not have an equivalent to U, the set of all keys for the objects, as present in the structure scheme.

$E =$ $\{(\text{lives at}, \{(\text{address}, \text{address})\}), (\text{p-isa}, \{(\text{SS\#}, \text{SS\#})\}, (\text{s-isa}, \{(\text{SS\#}, \text{SS\#})\}\}$
We again omit the renaming functions for the relations with only a single attribute: name, date and amount. In our example the renaming does nothing. But it is possible to use different names for the "same" attribute in two relations. For instance, we could have renamed the SS# for a person to "tax-id" in the professor relation.

$T =$ a complicated function which maps each object to a function from the set of possible primary-key values for the representation of an object to the set of real-world objects. Basically, this is the equivalent of the object world function, but now for the relational representation. If the "person" object with name "John Doe", and born on 1/1/1950 refers to a real person, then the tuple in the person-table with name "John Doe" and date of birth 1/1/1950 must refer to the same real person.

□

Note that the relational representation may involve the use of null values. This may happen when some functions are not total and it is guaranteed to happen when there are properties with mutually exclusive domains. In our example when a student has a loan, the scholarship-attribute must be null (and vice versa). These are so-called *non-existence* nulls, i.e. the value does not exist. (This in contrast to the more common interpretation of "value exists but is unknown")

Definition 2.4 *Conceptual Model*
A conceptual model is a pair consisting of a structure scheme and a representation scheme. □

Definition 2.5 *Database State*
For a conceptual model $\ll O, P, C, W >, < E, A, V, I, H, T \gg$ a *database state* is a function s, such that:

1. $\text{dom}(s) = O$.
2. $\forall o \in O : s(o) \subset \prod(V \restriction H(o))$.
3. There is a state graph g with $\text{dom}(g) = O \cup F$ and
$$\forall o \in O : g(o) = T(o)(s(o) \restriction\!\restriction I(o))$$
$$\forall f \in F : g(f) = \{< x, y > \in g(D(f)) \times W(R(f)) \mid$$
$$\exists z \in s(D(f)) : T(D(f))(z \restriction I(D(f))) = x \wedge T(R(f))(z \circ E(f)) = y\}.$$

□

Basically, this definition says that a database state is a (function mapping the set of names of object types to a) state graph and a corresponding set of relation instances (i.e. tables). By tying a database state to a state graph the constraints, expressed in the structure diagram, are imposed on the relational instances.

At any one time there is a one to one relationship between the database state s and the corresponding state graph g. We call g "the" state graph of s.

Definition 2.6 *State Space*
Given a conceptual model $\ll O, P, C, W >, < E, A, V, I, H, T \gg$, the *state space* S is the set of database states for that model. □

We can now define constraints on a conceptual model, and query and update database states.

3 The Data Language

Using the functional and the relational representation of a database we propose a new (formal) language for defining constraints, queries and updates:

Definition 3.1 *First Order Language*
Let $\ll O, P, C, W >, < E, A, V, I, H, T \gg$ be a conceptual model, with
$\mathbb{D} = \bigcup\{V(a) \mid a \in A\}$ and $\mathbb{T} = \bigcup\{\prod(V \restriction (u)) \mid u \subseteq A\}$.
Its *first order language* (FOL) L_F then consists of the following elements:

i) *Alphabet*
The alphabet is the union of the following sets of symbols

- constants: $\mathbb{D} \cup \mathbb{T} \cup O \cup F \cup A$
- variables: $\{X, Y, Z, X_1, Y_1, Z_1, \ldots\}$
- function symbols: $\{\circ, \cdot, ^{-1}, \mathrm{dom}, \mathrm{rng}\}$
- set symbols: $\{\cup, \cap, \setminus, \div, \bowtie\}$
- atom comparison symbols: $\{\leq, =, \geq\}$
- set comparison symbols: $\{\subset, =, \supset\}$
- tuple symbols: $\{\cup, \circ, \cdot, \mathrm{dom}\}$
- projection symbols: $\{\restriction, \prod\}$
- atom-set symbols: $\{\in\}$
- logical symbols: $\{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$
- quantors: $\{\forall, \exists, \$\}$
- punctuation symbols: $\{[,], \{,\}, (,), |, :, ;, ,\}$

ii) *Terms*

a-terms (defining a constant or the value of a tuple for an attribute)

- every $a \in \mathbb{D}$ is an a-term
- if t is a t-term and a an at-term then $t(a)$ is an a-term

at-terms (defining attributes)

- every $a \in A$ is an at-term
- if f is an af-term and a is an at-term then $f(a)$ is an at-term

t-terms (defining tuple constants and variables)

- every $t \in \mathbb{T}$ is a t-term
- every variable is a t-term
- if $a_1, \ldots, a_n$ ($n \in \{1, 2, \ldots\}$) are at-terms and $b_1, \ldots, b_n$ are a-terms then $\{(a_1; b_1), \ldots, (a_n; b_n)\}$ is an (enumerated) t-term
- if f is an f-term and t is a t-term then $f(t)$ is a t-term
- if t_1 and t_2 are t-terms then $t_1 \cup t_2$ is a t-term
- if t is a t-term and a an as-term then $t \restriction a$ is a t-term
- if f is an af-term and t is a t-term then $t \circ f$ is a t-term

s-terms (defining sets)

- every $o \in O$ is an s-term
- if $t_1, \ldots, t_n$ are t-terms then $\{t_1, \ldots, t_n\}$ is an (enumerated) s-term
- if f is an af-term and s is an s-term then $s \circ f$ is an s-term (this means overloading the $\circ$ operator so it applies to sets)
- if f is an f-term and t a t-term then $f^{-1}t$ is an s-term
- if f is an f-term and s is an s-term then $f(s)$ and $f^{-1}s$ 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 s is an s-term and a is an as-term then $\prod_a(s)$ is an s-term
- if X is a variable and s is an s-term and q a predicate then $\$[X : s \mid q]$ is an s-term
- if X is a variable, s is an s-term, q is a predicate and t is a t-term with at most X as a free variable then $\$[X : s \mid q \mid t]$ is an s-term
- if f is an f-term, then dom(f) and rng(f) are s-terms

f-terms (defining functions)

- every $f \in F$ is an f-term
- if $a_1, \ldots, a_n,\ b_1, \ldots, b_n$ are t-terms then $\{(a_1; b_1), \ldots, (a_n; b_n)\}$ is an (enumerated) f-term
- if f and g are f-terms then $f \circ g$ is an f-term
- if f is an f-term and s an s-term then $f \restriction s$ is an f-term

as-terms (defining sets of attributes)

- if $a_1, \ldots, a_n$ are at-terms then $\{a_1, \ldots, a_n\}$ is an (enumerated) as-term
- if s_1 and s_2 are as-terms and θ is a set symbol then $s_1 \theta s_2$ is an as-term
- if f is an af-term then dom(f) and rng(f) are as-terms
- if f is an af-term and s an as-term then $f(s)$ and $f^{-1}s$ are as-terms
- if t is a t-term then dom(t) is an as-term

af-terms (defining attribute functions)

- if $a_1, \ldots, a_n,\ b_1, \ldots, b_n$ are at-terms then $\{(a_1; b_1), \ldots, (a_n; b_n)\}$ is an (enumerated) af-term
- if f and g are af-terms then $f \circ g$ is an af-term
- if f is an af-term and s an as-term then $f \restriction s$ is an af-term
- if f is an af-term then f^{-1} is an af-term

terms

- a-, as-, at-, af-, f-, s- and t-terms are terms
- if t is a term then (t) is also a 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 t is a t-term and s an s-term then $t \in s$ is a predicate
- if s_1 and s_2 both are t-, f-, af-, as- or s-terms and θ a set comparison symbol, then $s_1 \theta s_2$ is a predicate
- if q_1 and q_2 are predicates and θ is a logical symbol, different from $\neg$ 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 with at most X as a free variable and s is an s-term then $\forall[X : s \mid q]$ and $\exists[X : s \mid q]$ are predicates
- there are no other predicates

□

In order to use this language we have to define the meaning (or interpretation) of the different terms and predicates. We do this both formally and informally.

Definition 3.2 *Interpretation*
Let $\ll O, P, C, W >, < E, A, V, I, H, T \gg$ be a conceptual model with first order language L_F. Let L_T be the set of terms without free variables and L_C the set of closed predicates. Furthermore, let S be the state space for this model. The *interpretation function* $\mathcal{I}_s$ satisfies[3]:

- $\mathcal{I} \in S \times (L_T \cup L_C) \rightarrow \mathbb{D} \cup \mathbb{T} \cup \mathcal{P}(\mathbb{T}) \cup (\mathbb{T} \rightarrow \mathbb{T}) \cup A \cup \mathcal{P}(A) \cup (A \rightarrow A) \cup \{\top, *, \bot\}$, where $*$ is the truth-value for undefined.
  This means that the interpretation of a term without free variables or a closed predicate for a given database state can be an element of the domain of an attribute or a tuple or a set of tuples or a function between tuples or an attribute or a set of attributes or a function between attributes or a (3-valued) truth-value. The interpretation of specific language constructs is described below.

- if $x \in \mathbb{D} \cup \mathbb{T} \cup A$ then $\mathcal{I}_s(x) = x$
  The interpretation of a value, tuple or attribute is itself.

- if t is a t-term and a an at-term then $\mathcal{I}_s(t(a)) = \mathcal{I}_s(t)(\mathcal{I}_s(a))$
  Given a tuple t and an attribute a, the interpretation of $t(a)$ is the interpretation of t applied to the interpretation of a.

- if f is an af-term and a an at-term then $\mathcal{I}_s(f(a)) = \mathcal{I}_s(f)(\mathcal{I}_s(a))$
  The interpretation of the application of an attribute-function to an attribute is straightforward (and yields an attribute).

[3] We have underlined the "terminal" symbols. Apart from the grammatical aspect of being terminal, this also means that these symbols take their normal mathematical meaning. Also, we sometimes use s to indicate an s-term. This s is not to be confused with the database state s, which occurs as suffix in $\mathcal{I}_s$.

- if $a_1, \ldots, a_n$ are at-terms and $b_1, \ldots, b_n$ are a-terms then
  $\mathcal{I}_s(\{(a_1; b_1), \ldots, (a_n; b_n)\}) = \underline{\{(}\ \mathcal{I}_s(a_1)\ \underline{;}\ \mathcal{I}_s(b_1)\ \underline{),\ldots,(}\ \mathcal{I}_s(a_n)\ \underline{;}\ \mathcal{I}_s(b_n)\ \underline{)\}}$
  Given a series of attributes and values, this is how we create a single tuple.

- if $o \in O$ then $\mathcal{I}_s(o) = s(o)$
  The interpretation of an object is its "value" in the database state. This value is a tuple in the table corresponding to the object's type.

- if $f \in F$ then $\mathcal{I}_s(f) = \underline{\{(}\ x\ \underline{;}\ y\ \underline{)}\ \mid x \in s(D(f)) \wedge y \in s(R(f)) \wedge$ for the state graph g of s the following holds: $< T(D(f))(x \restriction (I(D(f)))), T(R(f))(y \restriction (I(R(f)))) > \in g(f)\ \underline{\}}$
  The interpretation of a function from the structure scheme (i.e. a function from object types to object types) is a function in the representation scheme (i.e. a function from tuple types to tuple types) such that when an object in the state graph g for s is mapped to another object, the tuple in the database state, corresponding to the first object, is mapped to the tuple corresponding to the second object. (see Definition 2.5, item 3)

- if f is an f-term and t is a t-term then $\mathcal{I}_s(f(t)) = \mathcal{I}_s(f)(\mathcal{I}_s(t))$
  If f is a function between objects and t a tuple, then $f(t)$ is the corresponding function $\mathcal{I}_s(f)$ on tuples, applied to t.

- if t_1, t_2 are t-terms then $\mathcal{I}_s(t_1 \cup t_2) = \mathcal{I}_s(t_1) \underline{\cup} \mathcal{I}_s(t_2)$ if this is still a function, otherwise $*$.
  The interpretation of the union of tuples is the union (or concatenation) of their interpretations, but only if they have the same value for common attributes.

- if t is a t-term and a an as-term then $\mathcal{I}_s(t \restriction a) = \mathcal{I}_s(t) \underline{\restriction} \mathcal{I}_s(a)$
  This defines the projection of a tuple t onto the set of attributes a.

- if f is an af-term and t is a t-term then $\mathcal{I}_s(t \circ f) = \mathcal{I}_s(t) \underline{\circ} \mathcal{I}_s(f)$
  This defines the renaming of attributes in a tuple.

- if $t_1, \ldots, t_n$ are t-terms then $\mathcal{I}_s(\{t_1, \ldots, t_n\}) = \underline{\{}\mathcal{I}_s(t_1)\ \underline{,}\ \ldots\ \underline{,}\ \mathcal{I}_s(t_n)\underline{\}}$
  This defines (enumerated) sets of tuples.

- if f is an af-term and s is an s-term then $\mathcal{I}_s(s \circ f) = \underline{\{} t \underline{\circ} \mathcal{I}_s(f) \mid t \in \mathcal{I}_s(s)\underline{\}}$
  This generalizes functions on tuples to functions on sets of tuples.

- if f is an f-term and t a t-term then $\mathcal{I}_s(f^{-1}t) = \mathcal{I}_s(f)^{\underline{-1}}\mathcal{I}_s(t)$
  If we apply a function inversely to one tuple, we obtain a (possibly empty) set of tuples, i.e. an s-term. Note that this is a non-standard way of using $^{-1}$ as we are using non-injective functions, so f^{-1} is not a function any more.

- if f is an f-term and s an s-term then $\mathcal{I}_s(f(s)) = \underline{\{}\mathcal{I}_s(f)(t) \mid t \in \mathcal{I}_s(s)\underline{\}}$
  The interpretation of a function applied to a set of tuples is the set of tuples which are the result of applying the interpretation of f to the each of the tuples of s.

- if f is an f-term and s an s-term then $\mathcal{I}_s(f^{-1}s) = \underline{\cup\{}\mathcal{I}_s(f)^{\underline{-1}}(t) \mid t \underline{\in} \mathcal{I}_s(s)\underline{\}}$
  If we apply a function inversely on a set of tuples (which always contains tuples of only one tuple type), then the image is the set of inverse images of the tuples, i.e. an s-term. This again is a non-standard use of the $^{-1}$ operator.

- if s_1 and s_2 are s-terms and θ is a set symbol then $\mathcal{I}_s(s_1 \theta s_2) = \mathcal{I}_s(s_1)\ \underline{\theta}\ \mathcal{I}_s(s_2)$
  This means that the set symbols, applied to sets of tuples, take their normal meaning (in

the relational algebra). Note that this meaning can be undefined (i.e. $*$) if the sets are not "compatible" for the operation θ.

- if s is an s-term and a is an as-term then $\mathcal{I}_s(\prod_a(s)) = \underline{\{} t \underline{\restriction} \mathcal{I}_s(a) \mid t \in \mathcal{I}_s(s) \underline{\}}$
  This defines the projection of a set of tuples onto a set of attributes.

- if X is a variable, s an s-term, q a predicate with at most X as a free variable then $\mathcal{I}_s(\$[X : s \mid q]) = \underline{\{} \ y \in \mathcal{I}_s(s) \mid \mathcal{I}_s(q_y^X) = \top \underline{\}}$
  This defines the subset of those tuples of s that satisfy predicate q.

- if X is a variable, s an s-term, q a predicate and t a t-term with at most X as free variable then $\mathcal{I}_s(\$[X : s \mid q \mid t]) = \underline{\{}\mathcal{I}_s(t_y^X) \mid y \in \mathcal{I}_s(s) \wedge \mathcal{I}_s(q_y^X) = \top \underline{\}}$
  This defines a set of tuples, which is derived from the subset of those tuples y of s that satisfy predicate q, by interpreting the t-term t. (We will give an example later on to show the use of this construction.)

- if f is an f-term and $\theta \in \{\mathrm{dom}, \mathrm{rng}\}$ then $\mathcal{I}_s(\theta(f)) = \underline{\theta(}\mathcal{I}_s(f) \ \underline{)}$
  The domain and range of a function are sets of tuples.

- if $a_1, \ldots, a_n, b_1, \ldots, b_n$ are t-terms then $\mathcal{I}_s(\{(a_1; b_1), \ldots, (a_n; b_n)\})$ is $\underline{\{(}\mathcal{I}_s(a_1) \underline{;} \mathcal{I}_s(b_1)\underline{)}, \ldots, \underline{(}\mathcal{I}_s(a_n) \underline{;} \mathcal{I}_s(b_n)\underline{)\}}$
  This is an enumerated function between tuples.

- if f and g are f-terms then $\mathcal{I}_s(f \circ g) = \mathcal{I}_s(f) \underline{\circ} \mathcal{I}_s(g)$
  The $\circ$ operator keeps its mathematical meaning for concatenating functions.

- if f is an f-term, s an s-term then $\mathcal{I}_s(f \restriction s) = \mathcal{I}_s(f) \underline{\restriction} \mathcal{I}_s(s)$
  The $\restriction$ operator keeps its mathematical meaning for restricting the domain of a function.

- if $a_1, \ldots, a_n$ are at-terms then $\mathcal{I}_s(\{a_1, \ldots, a_n\}) = \underline{\{}\mathcal{I}_s(a_1) \underline{,} \ldots \underline{,} \mathcal{I}_s(a_n)\underline{\}}$
  This is how we define an enumerated set of attributes.

- if s_1 and s_2 are as-terms and θ is a set symbol then $\mathcal{I}_s(s_1 \theta s_2) = \mathcal{I}_s(s_1) \ \underline{\theta} \ \mathcal{I}_s(s_2)$
  The set symbols keep their mathematical meaning for sets of attributes.

- if f is an af-term and $\theta \in \{\mathrm{dom}, \mathrm{rng}\}$ then $\mathcal{I}_s(\theta(f)) = \underline{\theta(}\mathcal{I}_s(f) \ \underline{)}$
  The domain and range of a attribute-function are sets of attributes.

- if f is an af-term and s an as-term then $\mathcal{I}_s(f(s)) = \mathcal{I}_s(f) \ \underline{(}\mathcal{I}_s(s) \ \underline{)}$ and $\mathcal{I}_s(f^{-1}s) = \mathcal{I}_s(f)^{\underline{-1}}\mathcal{I}_s(s)$
  This is the straightforward generalization of the renaming of attributes (and its inverse) to sets of attributes.

- if t is a t-term then $\mathcal{I}_s(\mathrm{dom}(t)) = \underline{\mathrm{dom}}(\mathcal{I}_s(t))$
  The domain of a tuple is a set of attributes.

- if $a_1, \ldots, a_n, b_1, \ldots, b_n$ are at-terms then
  $\mathcal{I}_s(\{(a_1; b_1), \ldots, (a_n; b_n)\}) = \underline{\{(}\mathcal{I}_s(a_1) \underline{;} \mathcal{I}_s(b_1)\underline{)}, \ldots, \underline{(}\mathcal{I}_s(a_n) \underline{;} \mathcal{I}_s(b_n)\underline{)\}}$
  This defines an (enumerated) function on attributes.

- if f and g are af-terms then $\mathcal{I}_s(f \ \circ \ g) = \mathcal{I}_s(f) \ \underline{\circ} \ \mathcal{I}_s(g)$.
  The concatenation of two renamings is still a renaming.

- if f is an af-term and s an as-term then $\mathcal{I}_s(f \restriction s) = \mathcal{I}_s(f) \underline{\restriction} \mathcal{I}_s(s)$
  The restriction of a renaming to a subset of the attributes is still a renaming.

- if f is an af-term then $\mathcal{I}_s(f^{-1}) = \underline{\{(\, x \,;\, y \,) \mid (y;x) \in \mathcal{I}_s(f)\}}$.

- if t is a term then $\mathcal{I}_s(\ (t)\ ) = \mathcal{I}_s(t)$
  This says that parentheses have no meaning (other than to indicate grouping).

- if a_1 and a_2 are a-terms and θ is an atom comparison symbol then $\mathcal{I}_s(a_1 \theta a_2 = \mathcal{I}_s(a_1) \underline{\theta} \mathcal{I}_s(a_2)$.
  Depending on the "type" of the a-terms the operator θ may or may not be defined.

- if t is a t-term and s an s-term then $\mathcal{I}_s(t \in s) = \mathcal{I}_s(t) \underline{\in} \mathcal{I}_s(s)$

- if s_1 and s_2 both are t-, f-, af-, or s-terms and θ is a set comparison symbol, then $\mathcal{I}_s(s_1 \theta s_2) = \mathcal{I}_s(s_1) \underline{\theta} \mathcal{I}_s(s_2)$

- if q_1 and q_2 are predicates and θ is a logical symbol, different from $\neg$ then $\mathcal{I}_s(q_1 \theta q_2) = \mathcal{I}_s(q_1) \underline{\theta} \mathcal{I}_s(q_2)$

- if q is a predicate then $\mathcal{I}_s(\neg q) = \underline{\neg} \mathcal{I}_s(q)$

- if X is a variable, q a predicate and s an s-term then $\mathcal{I}_s(\underline{\forall}[X : s \mid q]) =$
  $\top$ if for all $y \in \mathcal{I}_s(s)$ we have $\mathcal{I}_s(q_y^X) = \top$,
  $\bot$ if there exists a $y \in \mathcal{I}_s(s)$, such that $\mathcal{I}_s(q_y^X) = \bot$, and $*$ otherwise.
  $\mathcal{I}_s(\underline{\exists}[X : s \mid q]) =$
  $\top$ if there exists a $y \in \mathcal{I}_s(s)$ such that $\mathcal{I}_s(q_y^X) = \top$,
  $\bot$ if for all a $y \in \mathcal{I}_s(s)$, we have $\mathcal{I}_s(q_y^X) = \bot$, and $*$ otherwise.

□

From a first order language, defined according to definition 3.1, we deduce the following 3 classes of constructs:

Definition 3.3 *Data Language*
Let $\ll O, P, C, W >, < E, A, V, I, H, T \gg$ be a conceptual model and let L_F be a first order language, where the alphabet is extended with the symbols ?, $\uparrow$, $\downarrow$ and ;. The data language L_D then contains L_F and the following elements:

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

ii) *queries*
Let X be a variable, let s be an s-term and let q be a predicate with at most X as free variable, then $?[X : s \mid q]$ is a query.
Let t be a t-term with at most X as free variable, then $?[X : s \mid q \mid t]$ is also a query. $?[X : s \mid q]$ is just a shorthand for $?[X : s \mid q \mid X]$. L_Q is the sublanguage of L_D, containing only queries.
Intuitively the construct $?[X : s \mid q \mid t]$ is analogous to the SQL construct
select t **from** s **where** q.
The construct $?[X : s \mid q]$ is simply **select** $*$ **from** s **where** q.

iii) *updates*

Let $o \in O$, $f \in F$, s_1, s_2 be s-terms, f_1, f_2 be f-terms without free variables, such that s_1 and f_1 both are enumerated, then: $o \uparrow s_1$, $o \downarrow s_2$, $f \uparrow f_1$, $f \downarrow f_2$ are updates.

$\uparrow$ indicates insertion, $\downarrow$ indicates deletion. Since the insertion adds information, not yet present in the database, the added information must be enumerated.

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.

□

Example 3.1 Recall the Student-Professor database given in Examples 2.1 and 2.2. Consider the following constraint: "A student who is also a professor cannot get a scholarship." In our data language we can write this as:

$$\forall[X : \text{student} \mid \text{s-isa}(X) \in \text{rng}(\text{p-isa}) \Rightarrow \neg(X \in \text{dom}(\text{scholarship}))]$$

This constraint is described using only elements of the structure scheme. Now consider the following constraint: "If a person is both a professor and a student then his empno must be the same as his studno."

$$\forall[S : \text{student} \mid \forall[P : \text{professor} \mid \text{s-isa}(S) = \text{p-isa}(P) \Rightarrow P(\text{empno}) = S(\text{studno})]]$$

Finally, consider the following query: "List the names of the professors, who are also a student and who (as a student) have a loan of at least 10.000, together with the amount of their loan."

$$?[X : \text{professor} \bowtie \text{dom}(\text{loan}) \mid X(\text{loan}) \geq 10.000 \mid X \restriction \{\text{name}, \text{loan}\}]$$

□

Definition 3.4 *Restricted State Space*
Let $\ll O, P, C, W >, < E, A, V, I, H, T \gg$ be a conceptual model with first order language L_F and interpretation function $\mathcal{I}$ as in definition 3.2, and let $SoC \subseteq L_C$ be a set of constraints then the *restricted state space* S_R is: $S_R = \{s \in S_f \mid \forall\ [\ q : SoC \mid \mathcal{I}_s(q) = \top]\}$ □

4 Future Research

We have enriched the data language for functional data models using the relational representation which exists for every functional structure scheme.

We want to investigate the expressive power of the new data language more extensively. A comparison to more purely functional languages such as Daplex [5] and to relational languages such as the relational algebra, tuple calculus or SQL [4] will be carried out. Also, a comparison with the logic-based language COL should be performed, as one can also express queries on both functional and relational schemes in COL [1].

Apart from a theoretical comparison of the expressive power of these languages it is also important to verify whether using relational representations will enable us to find short and easy formulations of queries, updates or constraints that are very difficult to describe using only the functional or only the relational model.

References

[1] Abiteboul S., S. Grumbach, COL: a logic-based language for complex objects. INRIA, France.

[2] Aerts A.T.M., K.M. van Hee, Modelleren met een Functioneel Datamodel. *Informatie* **31:12**, pp. 941–956, dec. 1989. (in Dutch)

[3] Codd E.F., A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM* **13:6**, pp. 377–387, June 1970.

[4] Paredaens J., P. De Bra, M. Gyssens, D. Van Gucht, The Structure of the Relational Database Model. *EATCS Monographs on Theoretical Computer Science* **17**, Springer Verlag, 1989.

[5] Shipman D., The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems* **6:1**, pp. 140–173, 1981.

[6] Ullman J.D., Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press, Rockville, MD, 1988.

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits.
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes.
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films.
85/04	T. Verhoeff H.M.L.J.Schols	Delay insensitive directed trace structures satisfy the foam the foam rubber wrapper postulate.
86/01	R. Koymans	Specifying message passing and real-time systems.
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specification of information systems.
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures.
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several systems.
86/05	J.L.G. Dietz K.M. van Hee	A framework for the conceptual modeling of discrete dynamic systems.
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP.
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers.
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun 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 responder's 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-1).

86/14	R. Koymans	Specifying passing systems requires extending temporal logic.
87/01	R. Gerth	On the existence of sound and complete axiomati zations 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 and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. 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	C. 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.

87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces.
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen.
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations.
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG.
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence.
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems.
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples.
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to Handle Structured Information.
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems.
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication.
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way.
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification.
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling.
88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits.
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases.
88/12	A.E. Eiben	Abstract theory of planning.
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees.
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types.

88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations.
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors.
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems.
88/18	K.M. van Hee P.M.P. Rambags	Discrete event systems: concepts and basic results.
88/19	D.K. Hammer K.M. van Hee	Fasering en documentatie in software engineering.
88/20	K.M. van Hee L. Somers M.Voorhoeve	EXSPECT, the functional part.
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 paralell 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, pp. 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. 23.

90/17 F.S. de Boer
C. Palamidessi

On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, p. 29.