LANGUAGE SUPPORT FOR OFFICE MODELLING

W. Lamersdorf[1],[2] G. Müller[2] J.W. Schmidt[3]

[1] Univ. Hamburg
FB Informatik
Schlüterstraße 70
D-2000 HAMBURG 13

[2] IBM Deutschland
Scientific Center
Tiergartenstraße 15
D-6900 HEIDELBERG

[3] Univ. Frankfurt
FB Informatik
Dantestraße 9
D-6000 FRANKFURT

ABSTRACT :

Office Modelling aims at providing abstract, conceptual tools for describing office system semantics in order to ease high-level, and more likely correct application programming. Advanced office systems are complex organizations of objects, locations, procedures, people, etc., combining independent workstations which cooperate concurrently in executing complicated office tasks.

In our approach to office modelling we perceive office procedures as being based on complex data object constructors that accept selected object components (e.g., addresses, dates, text-fragments) and return office objects of composite type (e.g., letters, forms, memos). We gain the semantic primitives required for *object construction*, *component selection*, and *type recognition* by generalizing the corresponding solutions provided by conventional record-based database models. This leads us from the 'flat structures' of traditional database models to 'recursive structures' allowing for flexible representations of non-formatted and highly related data objects as required for advanced office modelling.

1. FRAMEWORK FOR OFFICE MODELLING

An office is a complex organization of objects, procedures, locations, communications, and, last but not least, people. Advanced office organizations are, in general, distributed systems with independent workstations cooperating concurrently in executing office tasks as well as communicating with people. *Office Modelling* aims at providing abstract, formal tools to represent office semantics in order to ease high-level, and and more likely correct application programming. This paper introduces conceptual and linguistic tools to aid the development of advanced, computer based office information systems.

As "office work is, to a large extent, ultimately people work" [Elli83], computer support for advanced office systems has to concentrate on an appropriate subset of office activities. Office applications of particular interest for this paper are office information systems assisting various user in

- definition, selection, and manipulation of office information containing large amounts of not necessarily formatted *office data objects*, and

- distributed execution of various *office procedures* from user workstations which are nodes in a distributed office network.

In the following chapter we identify basic requirements for representing non-conventional information system applications (as, e.g., office systems), and then demonstrate how a 'recursive' data model can support them more adequately than traditional, record-based database modelling. The structural concepts for representing office data objects are extended in chapter three and applied to a major office example. Extended modelling tools for office procedures are proposed in chapter four. Specific aspects of data as well as procedure communication in a distributed office environment are, finally, addressed in chapter five.

2. MODELLING COMPOUND DATA OBJECTS

Modelling advanced data-intensive applications requires primitives for representing compound data objects adequately with respect to all their structural as well as operational aspects.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

An important criterion for the adequacy of a data model is the coherence of its modelling primitives with the underlying semantic structures of an intended class of applications. In office systems, for example, data objects to be specified are characterized by a great variety of structured components related by complex inner relationships.

2.1. SEMANTIC MODELLING PRIMITIVES

In our approach to office modelling we perceive office procedures as basically centered around semantic primitives allowing for

- *construction* of compound data objects out of elementary ones or those already constructed,
- *selection* of specific object components from compound data objects, and
- *recognition* of the construction rules (or types) underlying a compound data object instance.

For example, in an office system environment we want to be able to

- *construct* a 'letter' instance from components as, for example, names, addresses, dates, and textual fragments,
- *select* single letter components as, for instance, the sender's address or the letter's textual content, and
- *recognize* a given document as a letter (as opposed to a form, a memo, etc.).

Traditional database models as, for example, the relational data model [Codd70], support the semantic modelling primitives in a more limited way:

- 'Record' and 'relation' constructors allow for the representation of compound data objects of a fixed resp. variable length using content-based references (e.g. keys) for object identification and association. However, the gained flexibility ('relational relativism') has to be paid for by 'referential integrity' problems [Date81], [LaSc84].
- Component selection is well supported in cases where components can be represented by single tuples or attribute values. However, selection of more complex object components represented by sets of tuples spread over several relations requires complex query expressions that raise computational (relational 'completeness') and performance issues.
- Recognition of the construction of compound objects (i.e. types of relation expressions including the corresponding key constraints) is difficult or, in some cases, impossible [KSW81], [Klug80].

Limitations of traditional record-based data models [Kent79] become especially obvious when modelling data objects with highly varying structures which are semantically inter-related [Codd79] as required, for instance, in CAD/CAM [HaLo82], information retrieval [ScPi80], or office applications [GiTs83].

In comparison to the limited set of tools for conventional database modelling, data modelling in modern programming languages can be characterized by a great variety of modelling concepts (i.e. data type generators) which can be combined freely [Zill84]. The objective of the following 'recursive' data model is to integrate data abstraction mechanisms from modern programming languages with high-level data structuring, selection, and management concepts from current database technology.

2.2. RECURSIVE DATA MODEL PRIMITIVES

Recursive data models [Lame84] are based on simple as well as recursively defined data types [Hoar75] and extend the modelling tools of classical data models by allowing representations of *variable length* structured data objects, possibly nested in a varying depth.

For each recursively defined data type, RcsType, a recursive data model provides a set of different 'structure generators', Gen_i, to generate value instances of the respective recursive type. Structure generators are based on limited sets of component types, CompType_ij, which may, recursively, contain other recursive data types (including the data type to be defined), or consist of simple data types as known from conventional, high-level programming languages. Single components can be identified by elementary 'component selectors', sel_ij. Then,

```
TYPE RcsType =
        ( Gen_1 (sel_11: CompType_11; ...;
                 sel_1n: CompType_1n)
        | . . .  | Gen_i (...)  | . . .
        | Gen_k (sel_k1: CompType_k1; ...;
                 sel_km: CompType_km)      );
```

defines a data type whose value set consists of all hierarchically structured data values which can be generated by (in general nested) application of the generators 'Gen_i', 1≤i≤k, to components of type 'CompType_ij'. (Syntactic Note: In case of only a single structure generator, the outmost brackets of a recursive type definition may be omitted. Similarly, a single elementary component selector may be dropped.)

So, a recursively defined data type provides concepts for all three basic semantic modelling primitives:

- a set of *structure generators*, Gen_i, to construct a recursive value from a given set of component values (which are, in turn, either constructed by application of some structure generator, or are denoted by simple, conventional programming language expressions);

- for each structure generator a set of elementary *component selectors*, sel_ij, to select a component from a recursive variable whose value was generated by the generator 'Gen_i'. Syntactically, component selectors are enclosed in square brackets and written behind the variable to be selected. So, for a recursively defined variable, rcs_var, whose value was generated by 'Gen_i (...,comp_val_ij,...)',
  rcs_var [sel_ij] = comp_val_ij.
  Component selectors identify component *variables* (i.e., they may be used in an expression context, on the left-hand side of an assignment statement, as well as nested). Nested component variables are written one after another and evaluated from left to right.

- a set of Boolean *characteristic functions*, is-Gen_i, to recognize a recursive value as being constructed by some structure generator 'Gen_i'. For a recursive variable, rcs_var, for example, whose value was generated by application of the structure generator 'Gen_i (...)',
  is-Gen_i (rcs_var) = TRUE.

In terms of a recursive data model, an example 'document' type can now be defined as follows (for component type definitions see section 3.2):

```
DocumentType =
  ( Letter (sender,
            receiver: NameType;
            from, to: AddressType;
            date_mailed, date_received:
            DateType; content: TextType)
  | Form ( ... )
  | Memo ( ... )  |  . . .  );
```

Recursively defined data types comprise complex data values which are, in different ways, generated by (in general nested) applications of the structure generators defining the respective data types. Our recursive approach generalizes traditional (record-based) data models, restricted to data objects with an identical *structure* [Kent79], to more advanced data models that allow for the definition and manipulation of data objects sharing some *structuring concept*. In other words, recursive data models are based on a limited number of modelling primitives and gain their power by allowing for their orthogonal combination.

3. EXTENDED STRUCTURING TOOLS

In order to ease linguistic expressions for specific classes of office object semantics, we extend the recursive data model by a higher-level layer of abstract structural and operational modelling concepts.

3.1. ABSTRACT OBJECT REPRESENTATIONS

In general, recursive data structures are powerful enough to express all of the following data object representations. Subsequently, some additional data structuring mechanisms are introduced as short hand notations for those semantic structures which represent concepts occurring frequently in office system semantics.

3.1.1. Ordered Objects

The first important concept to be supported is that of an 'order' defined on application objects and leads to a 'list' type data structuring mechanism. Instead of defining a list type explicitly, as, for instance, by

```
ListType    =
   ( Empty ()
   | Append_list (last: ElementType;
                  rest: ListType)    );
ElementType = . . .;
```

we provide as a syntactically shorter, semantically equivalent alternative the predefined type definition for variable length lists:

```
ListType  =  LIST OF ElementType.
```

Single list instances are generated applying a typed standard generator for lists, 'ListType {...}'. (The syntax for complex value constructors follows that of a typed generator construct as, for example, in Modula-2 [Wirth82].)

Together with the 'list' type come the usual list operation primitives as, for instance, the element selectors 'LAST' (yields the most recently added element) and 'REST' (yields the list without the last element), 'list[i]' (selects the i-th element), 'ELEMS' (returns set of all list elements), 'LENGTH' (returns number of elements), 'VOID' (Boolean test for zero elements), ':+' (appends r.h.s. elements to l.h.s. list variable), etc.

For example, a document 'file' data type can now be modelled as

```
FileType  =  LIST OF DocumentType.
```

Then, for a corresponding 'file' variable, 'file[j]' selects the j-th document from a given file (if existent), and 'file [LAST]' selects the most recently added one; 'FileType {doc_1, ..., doc_n}' generates a file instance from given document instances, doc_1, ..., doc_n.

3.1.2. Object Ranges

Representing application object ranges leads to a 'set' type structuring concept. Sets have varying cardinality and no duplicate element values are allowed. The usual set operators (u, n, IN, :+, :-, etc.) are defined for set types which are specified syntactically as:

```
SetType  =  SET OF ElementType.
```

Set instances are generated applying a standard set generator 'SetType {...}'. Set restriction is based on first-order predicate expressions following the line of first-order query languages:

```
SetType { <element_id> IN <set>
                SUCH THAT <predicate> }
```

For example, a file cabinet may be perceived as a set of drawers:

```
FileCabinetType = SET OF DrawerType.
```

Then, 'cabinet_1 :+ cabinet_2' integrates into the set of drawers of a file cabinet variable, cabinet_1, the drawers of 'cabinet_2', and 'FileCabinetType {draw IN cabinet_1 SUCH THAT <predicate>}' selects a certain subset of drawers from 'cabinet_1'.

3.1.3. Object Identification

Object identification is an essential concept in office system applications and can be represented by 'mapping' object identifiers to the objects to be identified. Maps, in turn, are sets of pairs ε (Domain x Range) where 'Domain' and 'Range' are again sets, and no two element pairs have the same domain value; syntactically:

```
MapType = (DomainType ---> RangeType).
```

Map instances are created using a map generator 'MapType {d_1-->r_1, ..., d_n-->r_n}'. The operations on maps are 'DOM' (domain value set), 'RNG' (range value set), ':+' (map extension, i.e. inserting new pairs), ':&' (map update, i.e. overwriting existing pairs), 'map (argument)' (application of a map to an argument), etc.

For example, the 'DrawerType' referred to above can be modelled as a map from file identifiers to the corresponding files in the drawer:

```
DrawerType      = (FileIdType ---> FileType);
FileIdType      = TOKEN.
```

(File identifiers are regarded as unstructured 'tokens', only subject to equation tests.) Then, for example, 'drawer :+ DrawerType {file_i --> file}' extends the content of a given drawer by a new file which is identified by 'file_i'.

3.2. OFFICE OBJECTS EXAMPLE

In the following, we complete the example representation of a file cabinet containing files and documents as a subset of an office environment:

```
TYPE
 FileCabinetType = SET OF DrawerType;
 DrawerType      = (FileIdType --> FileType);
 FileType        = LIST OF DocumentType;
 DocumentType    = ( Letter (sender,
                       receiver : NameType;
                       from, to : AddressType;
                       date_mailed,
                       date_received : DateType;
                       content : TextType)
                   | Form (type_no, serial_no :
                       INTEGER; entries :
                       (EntryIdType-->EntryType))
                   | Memo (...)  |  . . .  );
```

Components of letters, forms, memos, etc. may again be defined recursively:

```
NameType         = Name (first, middle,
                     last : WordType);
AddressType      = Address (organization :
                     LIST OF WordType;
                     street, city, state,
                     country : WordType;
                     no, zip : INTEGER);
DateType         = Date (year: (1950..1990);
                     month : (1..12);
                     day   :  (1..31));
EntryType        = Entry (descr : TextType;
                     content : ContentType;
                     domain: SET OF ContentType);
ContentType      = . . .;
FileIdType       = TOKEN;
EntryIdType      = TOKEN;
```

A more complex structured data type is the 'TextType' comprising component types for paragraphs, sentences, words, etc.:

```
TextType         = LIST OF ParagraphType;
ParagraphType    = ( Titled_para  (title :
                       SentenceType;
                       cont : ParagraphType)
                   | Untitled_para (para :
                       LIST OF SentenceType));
```

```
SentenceType    = Sentence (elems :
                       LIST OF ElementType;
                       end_mark: ('.', '!',
                       '?', ...));
ElementType     = ( Word (WordType)
                  | Mark (',', ';', ...));
WordType        = LIST OF CHAR;
```

Finally, we are able to declare a single compound variable representing a file cabinet with all its structured content:

```
VAR file_cabinet : FileCabinetType.
```

3.3. DATA OBJECT INTEGRITY

In the proposed extended recursive data model, referential integrity problems are, in most cases, avoided by choosing appropriate data structuring primitives. In order to impose additional integrity constraints on data objects explicitly, Boolean 'invariant' predicates can be formulated with a data type definition. For a simple example, the following invariant restricts 'file' instances to containing not more than 99 documents:

```
INVARIANT FileType (file);
   CARD ELEMS file  ≤  99
END.
```

Object type invariants, however, have to be checked for validity after each manipulation of a corresponding object instance.

4. EXTENDED OPERATIONAL TOOLS

The basis for modelling office procedure semantics are the operational primitives of the extended object types in the proposed data model. They provided mechanisms for compound data object construction, decomposition, and recognition. More powerful operations on recursive objects can be defined (recursively) in terms of the elementary operators.

4.1. DATA OBJECT SELECTION

4.1.1. Elementary Selectors

For example, elementary selection of object components can be modelled in terms of the operational primitives as demonstrated in the following examples for a 'file_cabinet' variable as declared above:

"select the drawers from a file cabinet which contain a 'file_i'" :

```
DrawerType { draw IN file_cabinet
             SUCH THAT file_i IN  DOM draw }
```

"select the 4th document from 'file_i' in a given drawer" :

```
drawer (file_i) [4]
```

"is that document a form ?" :

```
is-Form (drawer (file_i) [4])
```

"select the content of the corresponding 'date' entry" :

```
drawer (file_i) [4] [entries] (date) [content]
```

A more complex example selects "the latest letter(s) from a sender 's' in a file 'f'" :

```
SET OF DocumentType { doc IN ELEMS f SUCH THAT
        (is-Letter (doc)
    AND (doc [sender] = s)
    AND  ALL doc' IN ELEMS f ((is-Letter (doc')
                      AND doc' [sender] = s) ==>
         (doc [date] ≥ doc' [date]))) }
```

4.1.2. Dedicated Selectors

Since component selection is frequent and, as shown in the last examples, sometimes complex in structured data object applications, it is supported additionally by 'dedicated component selectors'. Similar selector or view mechanisms have been proposed for the relational database model by [Ston75], [Rous82], and [MRS84], and are defined for the recursive data model in [Lame84].

For example, the previous component selector example can now be expressed recursively using a generic, parameterized 'selector' mechanism. The definition of the selector 'last_letter_from' is based on the fact that the latest letter was filed last (with unary list operators binding stronger than selectors):

```
SELECTOR last_letter_from FOR f: FileType
              (s: NameType) : DocumentType;
 IF VOID  f
    THEN <EXCEPTION: message ("There is
          no letter from", s, "in", f)>
    ELSE IF is-Letter (f [LAST])
              AND  f [LAST] [sender] = s
         THEN SELECT f [LAST]
         ELSE SELECTf [REST]
              [last_letter_from (s)]
END last_letter_from,
```

An application of this selector to file 'f', 'f [last_letter_from (s)]', selects either the required document or raises the defined exception.

4.2. DATA OBJECT MANIPULATION

Data objects are represented by variables as, for example, demonstrated in section 3.2. The simplest case of data object manipulation is to select an object or an object component, and then to replace it by a new value. For example, setting the receiving date of the last letter from a sender, s, in a given file to '1983-11-23' is:

```
   file [last_letter_from (s)] [date_received]
                               :=  Date (1983,11,23)
```

Insertion, deletion, or update of object (component) instances is expressible via set-oriented update operators and operands constructed by the extended operational primitives. For example:

"insert a file identified by 'file_i' into a given drawer" :

```
   drawer  :+  DrawerType {file_i --> file}
```

"delete some values from the domain of a given data entry" :

```
   entry [domain]  :-  SET OF ContentType
                       {value_1, ..., value_n}
```

"replace the last document in a file by a new one" :

```
   file [REST]  :+  new_document
```

More powerful query and data manipulation language constructs may include multi-component selection mechanisms as well as extended manipulation functions and control structures defined in terms of sets, lists, maps, etc. of object components.

4.3. OPERATION INTEGRITY

Integrity mechanisms could be provided to restrict procedure executions to states where certain 'pre- and/or post-conditions' hold (compare, e.g., 'dynamic consistency constraint' specifications in [BjJo78]). However, since in an appropriate programming language environment Boolean constraint checks can be specified anywhere in a procedure definition, the model does not support pre- and post-conditions explicitly.

An alternative approach to guaranteeing data object integrity is to restrict all operations manipulating instances of a data type to only the integrity preserving ones. This leads to the introduction of abstract data types (ADT) as utilized for databases by, for example, [Schm79]. ADTs can be supported by the data model, provided the use of structure generators can be restricted to ADT operations only. In the example above this means that a user may not simply append a document to a given file, but instead has to call a special 'file manipulation' function checking all consistency constraints.

5. SUPPORT FOR OFFICE COMMUNICATION

Representing advanced office applications with cooperating processes on different, distinct locations (as, e.g., outlined in [HKMS84]) requires additional support for modelling data as well as process communication.

5.1. DATA COMMUNICATION

In a first approach, we suppose that office procedures on different workstations communicate via sending data to globally accessible 'mailboxes' of respective peer work stations.

In such a case, our data model supports the representation of mailboxes by distinct global variables, and modelling data communication by communicating procedures accessing and changing these variables. The communicated data is represented by data objects as defined before. For example, if a global variable contains a 'mailbox' for each distinct workstation,

```
   TYPE MboxesType  = (WorkstationIdType
                                ---> FileType);
   VAR  Mailboxes    :  MboxesType;
```

a simple 'send' procedure could be user-defined as follows:

```
   PROCEDURE send (doc: DocumentType;
                   to : WorkstationIdType);
    VAR    mailbox    :   FileType;
    BEGIN mailbox    :=  Mailboxes (to);
          mailbox    :+  doc;
          Mailboxes :&  MboxesType
                            { to --> mailbox  }
    END send.
```

In order to avoid concurrency anomalies [EGLT76], shared variables have to be appropriately *locked* for access by more than one office procedure. Sophisticated locking strategies for parallel access to complex data objects can take advantage of the hierarchical structure of recursively defined data objects [GLP75].

Further extensions of the data model include a more general model for process communication and synchronization. Hoare's 'communicating sequential processes' (CSP) [Hoar78] provide the concepts for a non-deterministic communication scheme between concurrent processes coupled with process synchronization. The corresponding primitives for the office model are:

```
SEND <value> TO <workstation_id>; and
RECEIVE <variable> FROM <workstation_id>;
```

Different from CSP, however, in our model only 'receive' involves process synchronization (that is an implicit 'wait' for the message to be received), whereas 'send' is not synchronized but supported by an automatic buffering mechanism for messages sent but not yet received.

5.2. PROCESS COMMUNICATION

Besides communicating data, cooperating office procedures need inter-process communication in order to be able to synchronize their activities. So, execution of an office procedure on one workstation may either depend on the termination of a procedure on another workstation (control *convergence*), or may initiate a remote procedure on a different workstation which then executes concurrently (control *branch*). In our model, both kinds of synchronization primitives are supported, respectively, by a 'wait' and a 'trigger' construct.

```
TRIGGER <workstation> : <procedure-call>;
WAIT    <workstation> : <procedure-id>;
```

A 'trigger' activates execution of an office procedure on another workstation; a 'wait' delays a currently active procedure until a procedure on a different location has terminated execution.

Data communication between office procedures to be synchronized is either achieved via standard communication primitives (as defined above) or, in case of a control branch, via actual 'value' parameters passed to the triggered procedure.

For example, an office procedure processing an employee's travel application on his own workstation may cooperate, in various ways, with office procedures on other, remote office workstations:

```
PROCEDURE travel_application (form: DocumentType;
                 manager_ws, travel_office_ws,
                 archive_ws: WorkstationIdType);
VAR doc : DocumentType;
BEGIN
 ...<*fill out form*>...;
    form [entries] := ...;
 TRIGGER manager_ws : approve_form (form);
 ...<*other, concurrent activities*>...
 RECEIVE doc FROM manager_ws;
 IF is-Form (doc)
  AND doc [serial_no] = form [serial_no]
  THEN IF doc [entries] (manager_ok) = TRUE
       THEN BEGIN
         TRIGGER travel_office_ws : book (doc);
         WAIT travel_office_ws : book;
         SEND doc TO archive_ws  END
       ELSE ...<*application rejected*>...
  ELSE ...<*wrong form returned*>...
END travel_application;
```

Process synchronization may always lead to situations where a 'wait' condition never occurs, and, thus, the waiting procedure is delayed indefinitely. Deadlock detection or prevention mechanisms are only useful in special cases. In general, the data model has to provide concepts for expressing *exception handling* procedures to be executed after a certain 'time out' interval has passed in order to limit waits to a user-defined maximum.

In the 'travel_application' procedure above, for example, the 'wait' construct could be extended by the following exception handling mechanism:

```
WAIT travel_office_ws : book;
     FOR <*time_out_interval*>
     EXCEPTION TRIGGER travel_office_ws :
               reminder (doc); ... END;
```

6. CONCLUDING REMARKS

This paper is a first attempt to develop conceptual and linguistic tools to support formal representations of office system semantics and to ease high-level office application programming. The choice of the proposed data structuring mechanisms is influenced on the one hand by the relational approach to data management [Codd70], and, in particular, by our experience with relations in a programming language environment [Schm77]. On the other hand, our approach draws heavily from insights gained by applying 'META IV', the meta-language of the semantic specification method 'Vienna Development Method' (VDM) [BjJo78] in a database context (see, e.g., [BjLø82], [LaSc80]). Another approach to specifying data spaces based on recursive structures can be found in [CrHi82].

The paper concentrates on proposing conceptual tools for office data object and office procedure semantics modelling. The concepts are the kernel for an very high level office programming language as, for example, outlined in [HaKu80] or [SLTC82]. A complete office programming language can be derived by imbedding the proposed constructs into a conventional high-level programming language, or into an integrated database and programming language environment (e.g. [Schm84]).

In the examples sketched in the paper, the hosting programming language has an 'imperative', Pascal-like flavor. In general, however, the proposed concepts could also be adapted to a functional or logic-based programming language environment.

The basis for the proposed office data model is a recursive data model as introduced in [Lame84]. A prototype implementation of the recursive data model has been implemented at the University of Hamburg [Saun84], [LaSc84] utilizing a compiler writing system and a relational database environment. The prototype implementation maps our recursive approach to office object construction, selection, recognition, and manipulation down to the data objects and operations provided by the database programming language Pascal/R [ScMa80].

7. REFERENCES

[BjJo78] : D.Bjørner, C.B. Jones (Eds.) : "The Vienna Development Method: The Meta Language", Lecture Notes in Computer Science, no.61, Springer Verlag, Berlin Heidelberg New York, 1978.

[BjLø82] : D. Bjørner, H. Løvengreen: "Formalization of Database Systems - and a Formal Definition of IMS", Proc. 8th VLDB, Mexico City, September 1982, pp.334-347.

[BMS84] : M.L. Brodie, J. Mylopoulos, J.W. Schmidt (Eds.) : "On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages", Springer Verlag, Berlin Heidelberg New York, 1984.

[Codd79] : E.F. Codd : "Extending the Relational Database Model to Capture More Meaning", ACM TODS, vol.4, no.4, Dec. 1979, pp.397-434.

[Codd70] : E.F. Codd : "A Relational Model of Data for Large Shared Databanks", Comm. ACM, vol.13, no.6, June 1970, pp.377-387.

[CrHi80] : A.B. Cremers, T.N. Hibbard : "Specification of Data Spaces by Means of Context-free-grammar-controlled Primitive Recursion", Comp. Sci. Techn. Rep. no.107, Univ. Dortmund, 1980.

[Date81] : C.J. Date : "Referential Integrity", Proc. 7th Conference on VLDB, Cannes, France, September 1981, pp.2-12.

[Elli83] : C.A. Ellis : "Formal and Informal Models of Office Activity", in: R.E.A. Mason (Ed.): Proc. IFIP Congress 1983, Elsevier Science Publishers B.V. (North Holland), 1983, pp.11-22.

[EGLT81] : K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L.Traiger: "The Notions of Consistency and Predicate Locks in a Database System", Comm. ACM, vol.19, no.11, Nov. 1976.

[GiTs83] : S. Gibbs, D. Tsichritzis : "A Data Modelling Approach for Office Information Systems", ACM Transactions on Office Information Systems, vol.1, no.4, Oct. 1983, pp.299-319.

[GLP75] : J.N. Gray, R.A. Lorie, G.R. Putzolu : "Granularity of Locks in a Shared Data Base", Proc. 1st Conference on VLDB, New York, Sept. 1975, pp.428-451.

[HaKu80] : M.M. Hammer, J.S. Kunin : "Design Principles for an Office Specification Language", Proc. NCC 1980, AFIPS Press, 1980, pp.541-547.

[HaLo82] : R.L. Haskins, R.A. Lorie : "On Extending the Functions of a Relational Database System", Proc. ACM SIGMOD Int. Conf. on Management of Data, Orlando, Florida, June 1982, pp.207-212.

[Hoar78] : C.A.R. Hoare : "Communicating Sequential Processes", Comm. ACM, vol.21, no.8, August 1978, pp.666-677.

[Hoar75] : C.A.R. Hoare : "Recursive Data Structures", International Journal of Computer and Information Science, vol.4, no.2, 1975, pp.105-132.

[HKMS84] : R. Holliday, D. Kropp, G. Müller, W. Schulz : "Enduser Applications in Open Systems", IBM HDSC Techn. Report no. 84.06.007 IBM Heidelberg, June 1984.

[Kent79] : W. Kent : "Limitations of Record Based Information Models", ACM TODS, vol.4, no.1, March 1979, pp.107-131.

[Klug80] : A. Klug : "Calculating Constraints on Relational Expressions", ACM TODS, vol.5, no.3, Sept. 1980, pp.260-290.

[KSW81] : J. Koch, J.W. Schmidt, V. Wunderlich : "Type Derivation for First-Order Relational Expressions", Techn. Report no. 79/81, Fachbereich Informatik, Universität Hamburg, June 1981.

[Lame84] : W. Lamersdorf : "Recursive Data Models for Non-Conventional Database Applications", Computer Data Engineering Conference (COMPDEC), IEEE Computer Society, Los Angeles, April 1984.

[LaSc84] : W. Lamersdorf, J.W. Schmidt : "Specification and Prototyping of Data Model Semantics", Proc. Working Conf. on Prototyping, Namur, Belgium, Springer Verlag, Berlin Heidelberg New York, 1984.

[LaSc80] : W. Lamersdorf, J.W. Schmidt : "Specification of Pascal/R", Techn. Reports no. 73 and 74, Fachbereich Informatik, Universität Hamburg, July 1980.

[MRS84] : Mall, M. Reimer, J.W. Schmidt : "Data Selection, Sharing, and Access Control in a Relational Scenario", in [BMS84].

[Rous82] : N. Roussopoulos : "View Indexing in Relational Databases", ACM TODS, vol.7, no.2, June 1982, pp.258-290.

[Saun84] : L. Saunus : "Adaptive User Interfaces for Relational Systems Utilizing Compiler Writing Techniques" (in German), Diploma Thesis, Fachbereich Informatik, Universität Hamburg, 1984.

[ScPi82] : H.-J. Schek, P. Pistor : "Data Structures for an Integrated Database Management and Information Retrieval System", 8th Conference on VLDB, Mexico City, Sept. 1982, pp.197-207.

[Schm84] : J.W. Schmidt : "Database Programming: Language Constructs and Execution Models", in: U. Amman (Ed.): Proc. 8th GI Fachtagung on "Programming Languages and Program Development", ETH Zürich, Switzerland, Informatik Fachberichte vol.77, Springer Verlag, Berlin Heidelberg New York Tokio, 1984, pp.1-25.

[Schm79] : J.W.Schmidt : "Type Concepts for Database Definition", Proc. Intern. Conf. on: Databases - Improving Usability and Responsiveness, B. Shneiderman (Ed.), Haifa, August 1979.

[Schm77] : J.W. Schmidt : "Some High Level Language Constructs for Data of Type Relation", ACM TODS, vol.2, no.3, Sept. 1977, pp.247-261.

[ScMa80] : J.W. Schmidt, M. Mall : "Pascal/R - Report", Techn. Report no.66, Fachbereich Informatik, Universität Hamburg, West Germany, Jan. 1980.

[SLTC82] : N.C. Shu, V.Y. Lum, F.C. Tung, C.L. Chang : "Specification of Forms Processing and Business Procedures for Office Automation", IEEE Transactions on Software Eng., vol.8, no.5, 1982.

[Ston75] : M. Stonebraker : "Implementation of Integrity Constraints and Views by Query Modification", Proc. ACM SIGMOD Conf., 1975, pp.65-77.

[Wirt82] : N. Wirth : "Programming in Modula-2", Springer Verlag, Berlin Heidelberg New York Tokio, 1982.

[Zill84] : S.N. Zilles : "Types, Algebras, and Modelling", in: [BMS84], pp.441-450.