

Universität Ulm  
Fakultät für Informatik



Towards Customizable, Flexible  
Storage Structures  
for Complex Objects

Ullrich Keßler, Peter Dadam  
*Universität Ulm*

Nr. 93-07

Ulmer Informatik-Berichte

# **Towards Customizable, Flexible Storage Structures for Complex Objects**

Ullrich Keßler, Peter Dadam  
Universität Ulm  
Fakultät für Informatik  
Abt. Datenbanken und Informationssysteme  
89069 Ulm, Germany  
e-mail: {kessler, dadam}@informatik.uni-ulm.de


## **Abstract**

During the last years several new data models have been developed which directly support complex objects, especially their structure definition and manipulation. In most of these approaches the internal storage structures are directly derived from the logical structure of the complex objects following a fixed system inherent mapping strategy. The adequacy of a certain mapping strategy, however, is strongly dependent on the access patterns of the related applications. It should, therefore, be possible to define the internal representation (= physical schema) of complex objects irrespective of their logical structure (= external schema). In this paper we discuss appropriate base constructs for implementing complex objects and describe how to define alternative internal storage structures related to these base constructs by just specifying appropriate parameters in a respective data definition language. In addition, we also discuss how to control the clustering of these objects. Altogether, we achieve a flexibility which allows to define for a given logical complex object structure almost every storage and clustering structure discussed in the literature. Furthermore, many other variants and mixtures of storage structures not discussed yet can be defined.

## 1. Introduction

Because of the limitations of traditional data models to appropriately support engineering applications and so-called non-standard applications, several new data models have been developed during the last years. Typical examples are Smalltalk-like persistent object-oriented data models [Nier89], the C++ data model [Stro86], the molecule-atom data model [Mits88], the NF<sup>2</sup> data model [ScPi82], and its extension the extended NF<sup>2</sup> (eNF<sup>2</sup>) data model [PiAn86]. Based on these and other data models several DBMS systems and prototypes as for example O<sub>2</sub> [Banc88], AIM-P [Dada86], [DaLi89], Prima [HMMS87], Orion [Kim89], XSQL [Lori85], GemStone [MSOP86], DASDBS [Paul87], COCOON [ScSc90], [Scho92a], Ontos [Onto92], and ObjectStore [LLOW91] have been implemented. Opposed to traditional DBMS these systems directly support complex structured objects (complex objects, for short), especially their structure definition and manipulation.

In conjunction with these new data models and DBMS systems and prototypes also strategies to store these complex objects on page-oriented storage devices (disks) have been discussed. Respective solutions can be found in [Dada86], [DPS86], [HaOz88], [KFC90], [Kim89], [Lori85], [MSOP86], and [Sike88]. Most of these approaches decompose a complex object into records as the smallest low-level storage and access unit. Usually, these records are composed and stored using a fixed storage and clustering strategy. As one can easily show, there is no fixed decomposition strategy which is always the "best" for all possible applications and access patterns. If, for example, a complex object is stored in a small number of "large" records these "large" records must always be transferred into main memory, even if a query<sup>1</sup> accesses only a very small portion of the complex object. On the other hand, a strategy which always decomposes a complex object into many "small" records is also not a good solution in every case. Here, the retrieval of "small" portions of a complex object will be supported very well but the retrieval of complete substructures becomes very costly because of the large number of record accesses being necessary. Especially, if the path length of retrieving a record is very long or if the records are not well clustered these costs can sum up extremely. Therefore, a good decomposition strategy should store substructures which are often retrieved as a whole in just one or a few records and should separate those parts which are not frequently accessed together. In general, however, this aim cannot be achieved with a decomposition strategy which does not take the application profile into account. An example for this problem is illustrated by the following discussion on possible decompositions of an eNF<sup>2</sup> relation storing information on robots.

Fig. 1 shows a very simple example of an eNF<sup>2</sup> relation storing the construction data of robots like the geometry of their axes as well as some administration data like a report of the products being produced by each robot during the last weeks. If one follows the typical approaches mentioned above, in most cases each axis of each robot will be decomposed into 4 up to 8 records. However, if, in general, an axis is accessed as a whole a solution which stores a complete axis in just one record - as indicated in Fig. 1 by the -frame - might be much better because it may save many record accesses. On the other hand, in most cases the attributes "R\_nb", "Name", and "Instructions" will be stored together in one record even though they might better be stored in different records. This separation might save costs, if, for example, the name and the number of a robot are much more frequently accessed than the instructions. In this case the usually rather long instructions need not to be transferred into main memory every time.

---

<sup>1</sup> In this paper the notion "query" is used as a synonym for "query or data manipulation".

{Robots}										
R_nb	Name	<Axes>			{Production_report}			{Endeffectors}		Instructions
		Axis_nb	<Matrices>		Product	{Action}		E_nb	Function	
			Row	<Vector>		Week	Price			
R_1	Robi	1	1	<20,20,20,20>	box	17	200	E_1	screw	Robot ...
			2	<34,37,56,90>		30	300	E_2	weld	
			3	<21,34,78,60>	door	16	400	E_3	punch	
			4	<45,56,78,12>		29	100	E_4	drill	
		2	1	<16,90,30,14>		41	500			
			2	<16,42,45,78>						
			3	<12,79,59,78>						
			4	<23,67,31,67>						
R_2	Bigi	...	...	...	...	...	...	...	...	

**Description:** sets: {}, lists <>

user defined record structure:



user defined object independent record cluster:



**Fig. 1: Extension of an eNF<sup>2</sup> robots relation**

Another potential to optimize access behaviour is to cluster the records appropriately. In the context of complex objects, the most common heuristic is to try to store all records belonging to one complex object on adjacent pages ([BeDe89], [Dada86], [DPS86], [KFC90], [Kim87], [ScSi89]). With respect to our Robots relation this means that the records of each robot would be stored in a separate record cluster. However, if some portions of a complex object logically belong to the object but is seldom if ever accessed in conjunction with the rest of the object, this clustering strategy might have the opposite effect. If, for example, the production reports are frequently read for accounting purposes without needing the rest of the objects, a clustering strategy which stores all these reports in just one object independent cluster - as indicated in Fig. 1 by the shaded rectangle - might be much better.

In a system which offers just one physical representation for each logical construct, an application oriented "optimal" physical storage and cluster structure can only be achieved by changing the logical structure of the data. If this is done at some later point in time, all application programs accessing these objects have to be modified accordingly. A better solution would be to distinguish between the logical and the physical representation of a complex object. The idea of separating the logical data structure from its physical implementation is not new - in fact just the opposite. Such features are - to a certain degree - even already available in a number of commercial DBMS. In Ingres [Ingr90], for example, the user can choose among several storage structures for relations. In Oracle [Orac90] tuples of different relations can be clustered based on common values of their "clustering attributes". Other examples are the hierarchical database system IMS [Gee77] and CODASYL systems which offer storage structure description languages. However, in the context of complex objects there only exist a few projects which discuss how to define storage

structures and clustering strategies in dependency of the related applications. One of these projects is COCOON [Scho92a]. In the context of this project in the papers [Scho92], [TRSB93], and [RiSc93] several strategies to map networks of objects into NF<sup>2</sup> relations have been presented. Other proposals discussing how to realise application dependent object clusters can be found in [BeDe89], [Kim87], and [ScSi89].

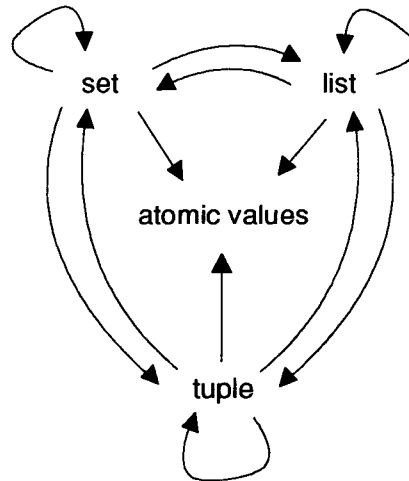
Using the eNF<sup>2</sup> data model as an example we will show that there exist in essence two degrees of freedom for defining storage structures for complex objects. The first one is to select data structures to implement sets, lists, and tuples. The second decision is whether to store the elements of these sets and lists or the attributes of the tuples, respectively, directly within these data structures or in referenced records. These two degrees of freedom can be controlled by just three orthogonal and simple parameters of an accordingly defined data definition language. By applying these parameters accordingly, the decomposition of a complex object can be defined almost arbitrarily. For example, nearly every storage structure discussed in [Dada86], [DPS86], [HaOz88], [KFC90], and [Lori85] can be realised. Even new variants, not discussed so far, can be defined as well. The set of possible structures is ranging from the one extreme where a complete complex object is stored in just one record to the other extreme where each atomic value is stored in a separate record. It is important to mention in this context that the logical structure of a complex object remains the same even if another storage structure is going to be used<sup>2</sup>. That is, application programs would not be affected by such changes to the physical structures.

Besides the decomposition of complex objects into (physical) records we will also discuss strategies, how to cluster the records in an application oriented fashion. These strategies allow to store nearly any subset of records in a separate cluster. That is, for almost any substructure of a complex object a separate cluster may be generated, if desired. On the other hand, so-called object independent clusters may be used to accumulate records of different complex objects within one and the same cluster. In the context of our Robots relation this means that for every robot a respective cluster may be generated to store the data for the axes and for the endeffectors in close neighborhood. At the same time, the production reports of all the robots may be collected altogether in a another (single) cluster.

Although we are using the eNF<sup>2</sup> data model here as the vehicle for our discussion, the results are not only applicable to systems which offer such a disjoint and non-recursive data model at the user interface, as for example AIM-P [Dada86] does. In DASDBS [Paul87], for example, a database kernel using NF<sup>2</sup> relations as an internal interface has been developed. On top of this kernel the COCOON data model [Scho92a] which is a recursive, non disjoint model has been implemented successfully. In other systems as O<sub>2</sub> [Banc88] or C++ like systems as ONTOS [Onto92] or ObjectStore [LLOW91] complex structures are build by lattices of objects. However, the objects themselves are hierarchically structured. Even in the case of flat objects, when complex objects are implemented only by references between these objects, proposals exists which discusses how to map these lattices onto hierarchical structures. [Kim87], for example, discuss how to define structures in Smalltalk-like systems like GemStone and Orion. Other papers as [BeDe89] and [ScSi89] introduce trees to represent object clusters in systems like O<sub>2</sub> and Prima. The reason is always that hierarchies can be clustered much easier than network-like structures.

---

<sup>2</sup> Changing the storage structure of an existing object may require an (internal) unload - reload operation or a respective catalogue management for dynamically handling such cases.



**Fig. 2: The eNF<sup>2</sup> data model**

The remainder of the paper is organised as follows: In Section 2 we briefly review the eNF<sup>2</sup> data model and introduce a data definition language which we will use in our further discussions. In Section 3 - the main part of this paper - we discuss the choices for the physical representation of sets, lists, and tuples using an appropriate data definition language. Section 4 describes how to define different clustering strategies for the different physical representations. In Section 5 we summarise the results and discuss our future research activities in this area.

## 2. The eNF<sup>2</sup> Data Model

For our further discussions we will use the eNF<sup>2</sup> data model as described in [PiAn86]. In this data model complex objects are composed of atomic values and, recursively applied, set, list, and tuple constructors (cf. Fig. 2). Typical atomic values are integer, real, and string. In this data model a complex object may be, for example, a "set of sets of strings", or just one simple "atomic value", or a much more complex structure as the Robot relation depicted in Fig. 1. Every relation in first normal form is also a legal object type of the eNF<sup>2</sup> data model.

To define the type and the storage structure of a complex object we will use a simplified data definition language because the basic principles of defining such storage structures are the main issue of this paper, not the syntactical constructs of such a language. The complete "language" used in this paper is given in Fig. 8 (see Appendix).

In this language the type of a typical employee relation (cf. Fig. 5) with the attributes "Emp\_nb", "Name", "Salary", and "Résumé" is defined as follows<sup>3</sup> (the storage structure of this relation will be defined later by filling the [...] brackets):

```

complex_object Employees [...]
    set [...] of tuple (Emp_nb [...]: integer,
                        Name   [...]: fix_string(30),
                        Salary  [...]: real,
                        Résumé  [...]: var_string)

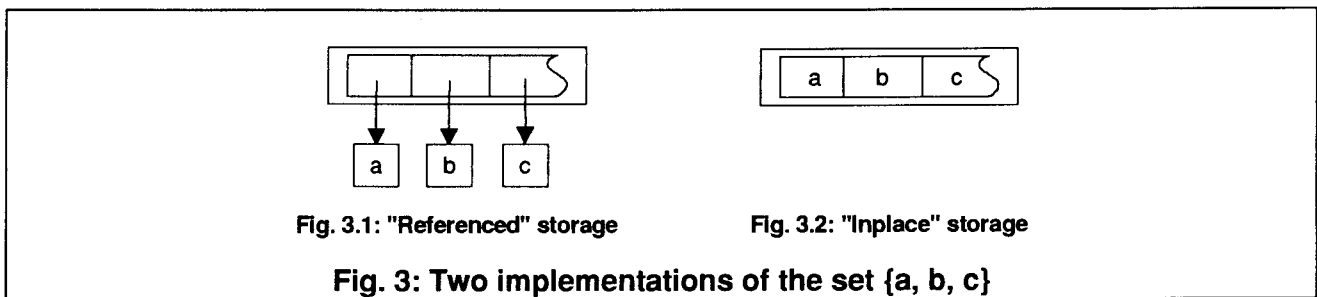
```

<sup>3</sup>To make the examples more readable key words are written in small letters and user defined names start with capitals.

### 3. Definition of Physical Storage Structures for Complex Objects

When defining physical storage structures for complex objects mainly two degrees of freedom exists. The first decision is to select appropriate data structures to implement sets, lists, and tuples. In the following, these internal data structures are called "constructor data structures". For example, the "constructor data structure" of a set or list may be an array of variable length or a linked list. The attributes of a tuple may be stored in one and the same record or may be distributed over several records.

The second decision is whether to store the elements of a set or list, or the attributes of a tuple directly within the constructor data structure or to store them in separate records. This is called an "inplace" or a "referenced" storage, respectively, of the elements or attributes. A very simple example is a set of atomic values which is implemented by an array of variable length. An "inplace" storage means that the values are directly stored within the array. A "referenced" storage means that each value is stored in an own referenced record. Both solutions are shown in Fig. 3 for the set "{a, b, c}".



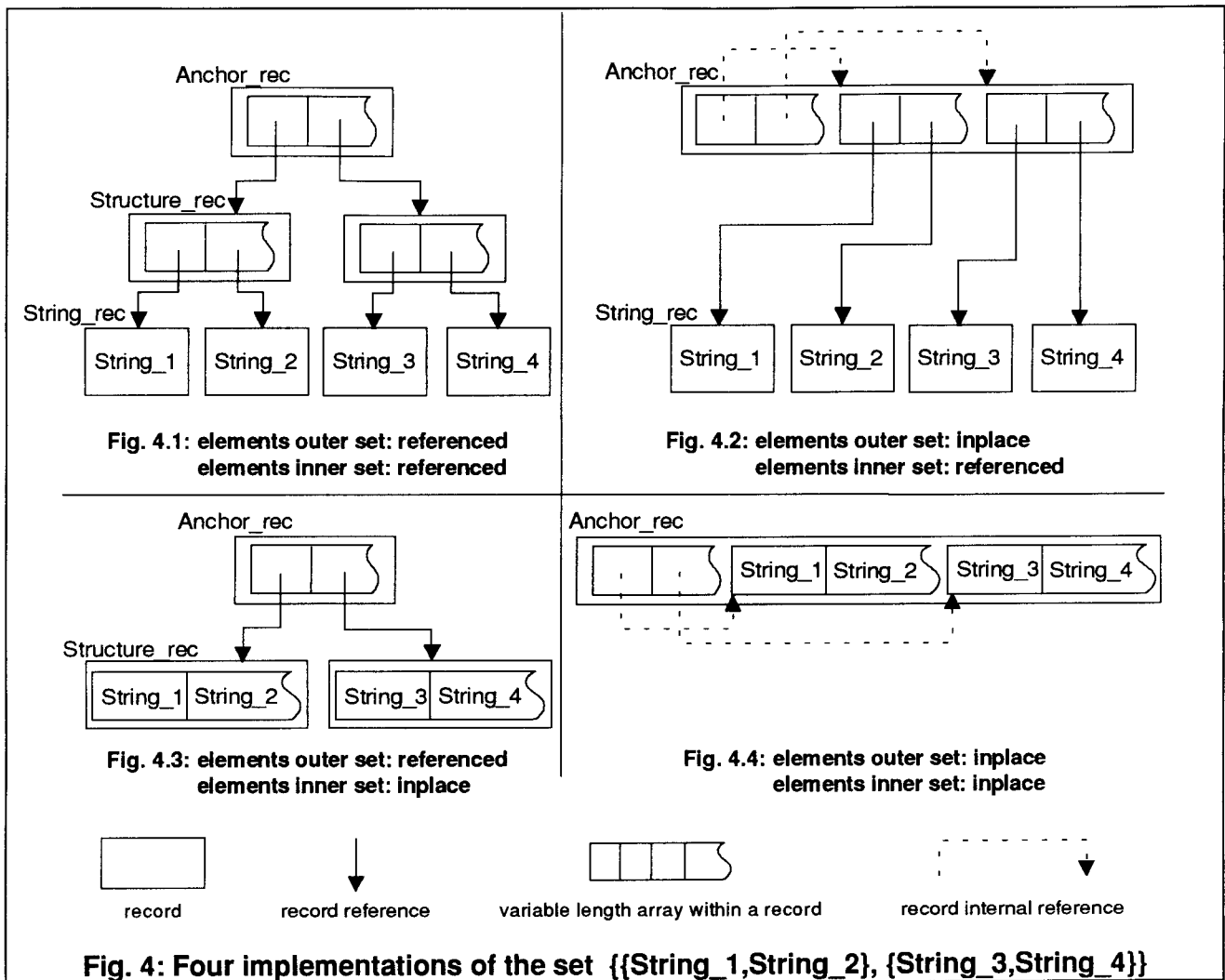
The decision whether to store elements or attributes "referenced" or "inplace" is not restricted to atomic values. Also complex structured elements or attributes may be stored "inplace" or "referenced". An example is the following set:

$\{\{\text{String\_1}, \text{String\_2}\}, \{\text{String\_3}, \text{String\_4}\}\}$ .

The elements "{String\_1, String\_2}" and "{String\_3, String\_4}" of the outer set are sets themselves. Therefore, these sets (= elements of the outer set) as well as their elements "String\_1", ..., "String\_4" (= elements of the inner sets) may be stored "inplace" or "referenced". The four resulting combinations and storage structures are depicted in Fig. 4<sup>4</sup>.

In Fig 4.1 it is assumed that the elements of the outer set as well as the elements of the inner sets are stored referenced. Therefore, the constructor data structure of the outer set contains references to records storing the constructor data structures of the inner sets. These constructor data structures themselves contain references to records with the values "String\_1", ..., "String\_4". In Fig. 4.2 the elements of the outer set are stored inplace but not those of the inner sets. Therefore, the constructor data structures of the two inner sets are stored within the same record as the constructor data structure of the outer set. The elements "String\_1", ..., "String\_4" of the inner sets are still stored in their own, separate records. Figs. 4.3 and 4.4 examine again the two cases in which the elements of the outer set are referenced or stored inplace, respectively. But now the elements of the inner sets are stored inplace in both cases. Therefore, the values "String\_1", ..., "String\_4" are stored directly within the constructor data structures of the two inner sets.

<sup>4</sup> It is assumed that for all sets only arrays of variable length are used as constructor data structures. If also linked lists are taken into account one gets at least 16 possible variants.



In the remainder of this section, after introducing the terms "record" and "record type name", we will discuss how to control both degrees of freedom - selection of a data structure and the decision whether to store elements or attributes inplace or referenced - by just three orthogonal parameters of an accordingly defined data definition language, as already mentioned in the introduction. These parameters will allow to define the internal representation of sets and lists as well as of tuples and, by doing so also of complete complex objects.

### 3.1 Records and Record Type Names

A "record" - as we use this term here - is a logical storage unit holding a variable number of bytes. Depending on the size of the records, several records may be stored within the same page but one record may also span several pages. Every complex object has one distinguished record serving as unique entry point. In the following this record is called "anchor record". Following the record references and the record internal references within this record leads to all components of the complex object.

In the following discussion and especially when defining the clustering of a complex object (see Section 4) it will be necessary to refer to classes of records by symbolic names. Therefore, we assume that records with semantically equivalent contents form so-called record types. When defining the storage structure of a complex object these record types get user defined unambiguous "record type names". For example, when defining the storage structure in Fig. 4.1 the record type names "Anchor\_rec", "Structure\_rec", and "String\_rec" may be assigned to the

record types as illustrated there. The record type name "Anchor\_rec" of the anchor record is specified in the parameter "anchor\_record\_type" (cf. Appendix, Fig. 8 [1]) of the data definition language used here. The two other record type names "Structure\_rec" and "String\_rec" are assigned in the type definitions of the corresponding sets.

### 3.2 Storage Structures for Sets and Lists

As already mentioned two independent degrees of freedom can be discriminated when defining the storage structures of sets and lists. The first one is the selection of a constructor data structure, the second one is the decision whether to store the elements inplace or referenced. Therefore, to control both degrees of freedom independently two parameters are needed. Consequently, we add two parameters - called "implementation" and "element\_placement" - to our data definition language (cf. also Appendix, Fig. 8, [2]-[7]):

```
object_type = ...
    /* Definition of a set. */
    set [implementation      = implementation_type,
        element_placement    = placement_type] of object_type |
    /* Definition of a list. */
    list [implementation     = implementation_type,
         element_placement    = placement_type] of object_type |
    ...
```

The parameter "implementation" is used to select the data structure to implement the set or list. If a system, for example, supports arrays of variable length and linked lists to implement sets and lists, this parameter would have two valid values (cf. Fig. 8, [11]):

```
implementation_type = array | linked_list
```

The second parameter called "element\_placement" is used to define whether the elements of the set are stored within the same record as the constructor data structure or within referenced records. Therefore, valid values of this parameter are (cf. Fig. 8, [10]):

```
placement_type = inplace | referenced (record_type_name)
```

If the value "inplace" is used, the elements will be stored within the same record as the constructor data structure (cf. the elements "String\_1", ..., "String\_4" in Figs 4.3 and 4.4). On the other hand, if the value "referenced" is used each element of the set or list will be stored in its own "referenced" record. These "referenced" records are forming an own record type. The unambiguous name of it must be given in the parameter "record\_type\_name" (in Figs. 4.1 and 4.2, for example, the elements "String\_1", ..., "String\_4" are stored in referenced records of type "String\_rec").

To demonstrate the usage of these parameters we will discuss now how to define the different storage structures in Fig. 4. The complete definition of the structure in Fig. 4.1 is:

```
complex_object Set_of_set_of_strings [anchor_record_type = Anchor_rec]      1
set [implementation=array, element_placement=referenced (Structure_rec)A] of 2
    set [implementation=array, element_placement=referenced (String_rec)B] of 3
    var_string.                                                                4
```

In line 1 the name "Set\_of\_set\_of\_strings" of the complex object and the name "Anchor\_rec" of the type of the anchor record are declared. Line 2 expresses that the complex object is a set and an array has to be used for its implementation. The elements of the set shall be stored in referenced records. The name of the record type being formed by these records shall be

"Structure\_rec". Line 3 defines that the elements of this set are themselves sets which are also implemented by arrays. The elements of these inner sets are stored in referenced records forming the type "String\_rec". Finally line 4 defines that the elements of these inner sets are strings of variable length.

Having once defined the storage structure of Fig. 4.1, the other three storage structures in Figs. 4.2, 4.3, and 4.4 can be obtained by just changing the values of the parameters "A" and "B", respectively. To define the storage structure in Fig. 4.2 the value of the parameter "A" must be set to "element\_placement = inplace", the parameter "B" remains unchanged. On the other hand, to define the storage structure in Fig. 4.3 the parameter "B" must be set to "element\_placement = inplace" and the parameter "A" remains unchanged. Last but not least, to get the structure of Fig. 4.4 which stores the complete complex object in just one record both parameters "A" and "B" must be set to "element\_placement = inplace". Please note, that we have got the four different storage structures by just changing the parameters "A" and "B". If both "implementation" parameters are also subject to change (e.g. "implementation = linked\_list") we obtain at least 16 different storage structures to implement the set of set of strings.

### 3.3 Storage Structures for Tuples

Having discussed the representation of sets and lists in detail we will now turn over to storage structures for tuples. In principle, the same degrees of freedom - definition of a constructor data structure, decision whether to store the attributes inplace or referenced - are appearing here, too. In this case the first degree of freedom corresponds to the decision whether to store a tuple - or more precisely its constructor data structure - in just one record or to distribute it over several records (= vertical partitioning of a relation). The second degree of freedom is the decision whether the attribute values are stored within the same records as the constructor data structure or in referenced records. To control both degrees of freedom independently again two parameters are needed. Therefore, we introduce the parameters "location" and "element\_placement" (which has the same meaning as in sets and lists) into the term to define attributes (cf. Fig. 8, [8]-[9]):

```
attribute_description = attribute_name [location = location_type,
                                     element_placement=placement_type]: object_type
```

Our mental model of a constructor data structure for a tuple is very similar to a record in a PASCAL-like programming language. For each attribute of a tuple one field is reserved within the constructor data structure. Whether this field contains the attribute value itself (= inplace storage) or a reference to a record which stores the attribute value (= referenced storage) is controlled by the parameter "element\_placement". As this parameter is bound to the attribute definition, the decision whether to store an attribute value<sup>5</sup> inplace or referenced can be made independently for each attribute.

Sometimes, if, for example, some attributes are accessed very seldom it may be useful to distribute a tuple over several records. Therefore a mechanism is provided that allows to divide the constructor data structure of a tuple into one primary and several secondary blocks. Each of these blocks may contain one or more fields of the constructor data structure. Each secondary block is stored in a separate record. The references to these records are also stored (in addition to the fields for normal attributes) in the primary block.

---

<sup>5</sup> If the attribute value itself is a complex structured subobject the corresponding constructor data structure of the attribute value is stored inplace or referenced.

Employees			
Emp_nb	Name	Salary	Résumé
77234	Johns	4000	Mrs. Johns is born ...
77235	Smith	4400	Mr. Smith is born ...

**Fig. 5: Extension of an employees relation**

To define whether an attribute (resp. its corresponding field) is located in the primary block or in a secondary block the parameter "location" is used. This parameter can take one of the following two legal values (cf. Fig. 8, [12]):

`location_type = primary | secondary (record_type_name)`

If the value "primary" is specified for an attribute its corresponding field is located in the primary block. If, however, the value "secondary (record\_type\_name)" is used, the field is located in a secondary block. This secondary block is stored in a record of type "record\_type\_name". If more than one attribute shall be stored in the same secondary block just the same record type name has to be used in the parameter "record\_type\_name" for all these attributes.

The interaction of both parameters "location" and "element\_placement" shall be demonstrated now using the Employees relation of Fig. 5 as an example. We assume that the attributes "Salary" and "Résumé" are accessed very seldom. Therefore, the attributes "Emp\_nb" and "Name" shall be stored in the primary block while the attributes "Salary" and "Résumé" shall be stored together in a secondary block. As the résumé itself may be very long its value shall be stored in a referenced record. This example shall also be used to demonstrate how to implement a relation by a linked list. A possible definition satisfying all these requirements is:

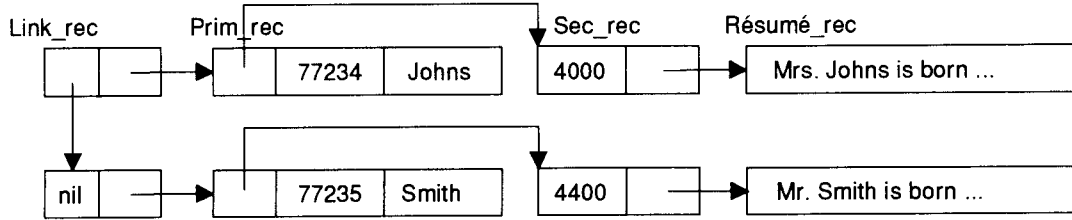
```
complex_object Employees [anchor_record_type=Link_rec]
set [implementation=linked_list, element_placement=referenced (Prim_rec)]
of tuple (Emp_nb [location=primary, element_placement=inplace]: integer,
         Name   [location=primary, element_placement=inplace]: fix_string(30),
         Salary [location=secondary (Sec_rec), element_placement=inplace]: real,
         Résumé [location=secondary (Sec_rec),
                  element_placement=referenced(Résumé_rec)]: var_string).
```

The resulting storage structure is illustrated in Fig. 6.1.

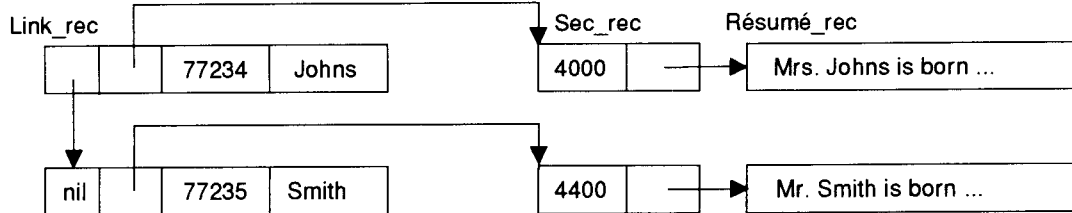
This storage structure is a good example why both parameters "location" and "element\_placement" are needed. If one of them would be omitted it would be impossible to express that the reference to the résumé record shall be stored in the same secondary record as the value of the salary. It would only be possible to store the reference of the résumé record within the primary block.

A drawback of the storage structure shown in Fig. 6.1 is that there are many small link records, each of it used to store one pointer to the primary block of the tuple and another pointer to the next tuple. This problem, however, can be avoided very easily by storing the primary blocks of the tuples as shown in Fig. 6.2 within the link records. This is expressed by changing the value of the parameter "element\_placement" in the second row of the definition from "referenced (Prim\_rec)" to "inplace":

```
complex_object Employees [anchor_record_type=Link_rec]
set [implementation=linked_list, element_placement=inplace] of ...
```



**Fig. 6.1 Tuples with referenced primary blocks**



**Fig. 6.2 Tuples with primary blocks stored inplace**

**Fig. 6: Two possible storage structures for the employees relation**

### 3.4 An Example of a Storage Structure for the Robots Relation

The discussion of the two degrees of freedom - selection of a constructor data structure and decision whether to store elements and attributes inplace or referenced - as well as of the parameters "element\_placement", "implementation", and "location" to control these degrees of freedom is now complete. Fig. 9 shows a complete type definition for the eNF<sup>2</sup> Robots relation (cf. Fig. 1) using one possible physical storage structure. In this rather comprehensive example, all the constructs and parameters introduced so far are used in conjunction. The selected storage structure follows the assumptions made in the introduction. Those parts of the relation being accessed often as a whole are stored together in the same records and those parts not frequently accessed together are separated. The resulting storage structure is depicted in Fig. 10.

The following details of the storage structure may be worthwhile to be mentioned. The constructor data structure of each Robot tuple has been distributed among the primary block and one secondary block. In the definition of the Robots relation (Fig. 9) this is expressed by the lines [1], [2], [3], [13], [16], and [18]. This decomposition allows (see Section 4 for details) to create one record cluster for each robot and store the data of the axes and endeffectors in these clusters object wise. At the same time, the data of all production reports of all robots can be stored together in one single object independent record cluster. Another feature of the storage structure is the representation of the axes. The complete data of an axis is stored in a single "Axis" record (cf. Fig. 9: [5] - [12]). To implement the lists of axes variable length arrays are used (cf. Fig. 9: [4]). These variable length arrays are stored inplace of the "Sec\_rec" records (Fig. 9: [3]). This kind of implementation of a list (or a set) corresponds to the DASDBS (cf. [DPS86]) implementation of complex objects. On the other hand, the implementation of the production reports follows the strategy of AIM-P (cf. [Dada86]) to implement sets and lists. A variable length pointer record is referenced by a field of the primary block of a robot tuple (cf. Fig. 9: [13]). This pointer record carries the references to the data records storing the names of the products being worked on and to records storing the amount of work (cf. Fig. 9: [14], [15]). Finally, the implementation of the sets of endeffectors corresponds to the ideas of XSQL (cf. [Lori85]). The elements of the sets are linked together by twin pointers (cf. Fig. 9: [17]). The usually rather long instructions of the robots are stored in separate records (cf. Fig. 9: [18]).

## 4. User Defined Clustering of Complex Objects

The methods discussed in Section 3 allow to define application oriented decompositions of complex objects into appropriate record structures. However, nothing has been said about the physical placement of these records on disk so far. In the following we will, therefore, discuss mechanisms which ensure that records being accessed frequently together are stored on the same or on adjacent pages to reduce the number of page faults when accessing them. As in the previous discussion of storage structures a "hard-wired" clustering strategy as, for example, to store always the records of one complex object on adjacent pages may be good in many (perhaps even most) cases but may also fail sometimes. In these cases the user should be able to change the clustering strategy accordingly. In the following we will first introduce the logical terms "segment" and "record cluster" and then discuss how the distribution of records among these clusters and segments can be controlled.

### 4.1 Segments and Record Clusters

In the following, we assume that a "segment" is a linear, page oriented address space having a unique segment identifier. In a file based system a segment is usually mapped to one or more direct access files. Each segment contains one or more "record clusters", each of which storing an arbitrary large number of records of various types. The system will attempt to store records which belong to the same cluster on adjacent pages of a segment. The number of record clusters within a segment is not static but can grow or shrink as related objects or subobjects are inserted into or deleted from the database. This is very similar to the kind of clusters being used in the commercial database system Oracle [Orac90].

### 4.2 Object-Centred and Object-Independent Record Clusters

As already indicated in the introduction two kinds of record clusters can be discriminated. One kind of record clusters are used to collect records belonging to a specific object or subobject. Because of this behaviour we will call them "object-centred" record clusters in the sequel. They compare to the "classic" clusters as discussed, for example, in [BeDe89], [Dada86], [DPS86], [KFC90], [Kim87], [ScSi89]. The second kind of record clusters are used to store records independently of their object membership in the same record cluster. We, therefore, will call them "object-independent" record clusters.

To define which records of an object are stored in object-centred and which are stored in object-independent clusters we use the syntax given in Fig. 7. The respective specification is based on the record types rather than the record instances because it makes query optimisation rather difficult, if the clustering structure has to be determined at instance level at run-time.

#### 4.2.1 Object-Centred Record Clusters

To express which records are stored in object-centred record clusters we use so-called "object-centred record cluster types". Each object-centred record cluster is an instance of such a cluster type. To identify the "correct" instance when storing a record we associate with each object-centred record cluster a so-called "identifying record"<sup>6</sup> which serves as the "identifier" of the

---

<sup>6</sup>Because an "identifying record" serves only (via a system internal identifier) as the identifier of an object-centred record cluster, it is not necessary that the identifying record itself is stored in the cluster.

```

cluster_definition = /* Definition of an object-centred record cluster type. */
                    object_cluster_type (cluster_type_name = cluster_type_name,
                                         segment           = segment_name,
                                         identifying_records = record_type_name,
                                         member_records     = list_of_record_types) |

                    /* Definition of an object-independent record cluster. */
                    segment_cluster (cluster_name = cluster_name,
                                     segment       = segment_name,
                                     member_records = list_of_record_types)

                    /* List of "member" record types of a record cluster. */
list_of_record_types = {record_type_name {, record_type_name}*)

cluster_type_name = string /* Name of an object-centred cluster type. */
cluster_name      = string /* Name of an object-independent record cluster. */
segment_name      = string /* Name of a segment. */

```

**Fig. 7: Term to define object-centred and object-independent record clusters**


cluster. Such a cluster type and the identifying records are defined by the first variant of the term for defining clusters (cf. Fig. 7):

```

cluster_definition = object_cluster_type (cluster_type_name = cluster_type_name,
                                         segment           = segment_name,
                                         identifying_records = record_type_name,
                                         member_records     = list_of_record_types)

```

This declaration reads as follows: An object-centred cluster type called "cluster\_type\_name" is defined. The instances of this cluster type are stored in the segment specified by "segment\_name". Their identifying records are of the type given in the parameter "identifying\_records". This implies that for each record of this type one instance of the cluster type is created. The "member\_records" clause specifies which records of which record types will be stored in these clusters. Identifying records and their related member records either have to be in a (may be transitive) parent - child relationship or have to be the same type.

This process of defining object-centred record clusters shall be illustrated by an example. Again, we use the Robots relation and follow the assumptions in the introduction that all data of the axes and endeffectors shall be collected in an own, separate record cluster for each robot. This means that for each robot one record cluster is needed which stores the "Sec\_rec", "Axis\_rec", and "Endeff\_rec" records as depicted in Fig. 10 by the -shade. The definition of the corresponding object-centred record cluster type is:

```

object_cluster_type (cluster_type_name = Sec_rec_cluster,
                    segment             = Main,
                    identifying_records = Sec_rec,
                    member_records      = (Sec_rec, Axis_rec, Endeff_rec))

```

This definition means that for each record instance of type "Sec\_rec" one record cluster is created in the segment "Main". In Fig. 10 these clusters are labelled with "Sec\_rec\_cluster-1" and "Sec\_rec\_cluster-2". Depending on their hierarchical dependency of the identifying "Sec\_rec" records all "Sec\_rec", "Axis\_rec", and "Endeff\_rec" records are stored within these two clusters.


It should be mentioned that in this process any record type of any hierarchical level may be used as the identifying record type to define an object-centred cluster type. This allows to define a large variety of clustering strategies for the substructures. Furthermore it is not necessary (as mentioned in footnote 6) that the identifying records themselves are stored in the record clusters which they generate. This is shown by the following example:

```
object_cluster_type (cluster_type_name    = Prim_rec_cluster,
                    segment                = Secondary,
                    identifying_records    = Prim_rec,
                    member_records        = (Inst_rec))
```

This definition says that the "Inst\_rec" records which store the instructions of the robots are stored in object-centred record clusters which are localised in the segment "Secondary" (cf. Fig. 10: "Prim\_rec\_cluster-1" and "-2"). However, the "Prim\_rec" records themselves which are the identifying records of these record clusters are stored in an object-independent record cluster as defined in the following section.

#### 4.2.2 Object-Independent Record Clusters

Object-independent record clusters are used to cluster records independently of their object membership. In contrast to object-centred record clusters only one extension of an object-independent record cluster (type) exists in this case. Therefore, these record clusters may be identified by unambiguous user defined names. As there do never exist two object-independent record clusters with the same name in a segment, we call these clusters also "segment clusters" (cf. Fig. 7). Their usage shall be demonstrated also using again the Robots relation as an example.

In the introduction we assumed that the production reports are frequently accessed without needing the other parts of the object structure. Therefore it may be favourable to store all records of the types "Pointer\_rec", "Product\_rec", and "Action\_rec" together in one object-independent record cluster. This is expressed by the following term. Its effect is that all records of this three types are stored in the "Costs\_cluster" (cf. the -shade in Fig. 10) which is localised in the segment "Main".

```
segment_cluster (cluster_name    = Costs_cluster,
                segment          = Main,
                member_records    = (Pointer_rec, Product_rec, Action_rec))
```

Another example are the "Anchor\_rec" and "Prim\_rec" records. These records are involved in all kinds of object access. Therefore it seems useful to collect all these records independently of their object membership in a single cluster. In Fig. 10 this cluster is labelled "Anchor\_cluster" and is defined as follows:

```
segment_cluster (cluster_name    = Anchor_cluster,
                segment          = Main,
                member_records    = (Anchor_rec, Prim_rec))
```

The last term completes the cluster definitions for the Robots relation. In the structure discussed, the data of the axes, endeffectors, and instructions are clustered in a "classical" object-centred manner while the data of the production reports are collected object-independently in a single cluster. This potential to define a mixture of object-centred and object-independent clusters distinguishes this approach from others which support only object-centred clusters.

## 5. Conclusion and Future Work

The central issue of this paper is the description of a basic mechanism which allows to separate the logical structure of complex objects from their physical implementation. By doing so, a large variety of different physical storage structures can be used to implement a given logical structure. In the first part of this paper (Section 3) we discussed how to define application oriented system-internal storage structures for complex objects. We elaborated two degrees of freedom for defining storage structures for complex objects. The first one is the selection of appropriate constructor data structures to implement sets, lists, and tuples. The second one is the decision whether to store elements of sets and lists and attributes of tuples directly within of the constructor data structures or in referenced records. To control both degrees of freedom independently we have introduced three orthogonal parameters. These parameters allow to define a large variety of different physical structures for complex objects. Using this approach, the selection of a "good" or "bad" storage structure influences only the access performance but not the logical behaviour of a complex object. This guarantees that application programs need not to be changed if the storage structure shall be changed for performance reasons at some later point in time.

The definition of a good storage structure, however, is only the first step towards an optimized storage of complex objects. We, therefore, discussed in the second part of the paper (Section 4) also new strategies for clustering the records on disk. Otherwise one might have a good storage structure but each record access may cause a page fault. Therefore, we elaborated two kinds of clusters and described how to define them. The first kind of clusters are so-called object-centred clusters which collect records of objects or subobjects in a "classical" object-centred manner. The second kind of clusters are so-called object-independent record clusters which store records independently of their object membership. The combination of these cluster types and the possibility to define more than one object-centred cluster for each object or subobject allows to define a large variety of different clustering strategies. Taking both approaches together, it is no longer necessary to choose an inadequate logical complex object structure just for performance reasons.

However, the definition of an optimized storage and clustering structure does not automatically guarantee that an optimal access performance is achieved. In addition, also the process of query optimisation must be adjusted accordingly to use these storage structures optimally. In [KeDa91] we have already discussed various strategies to use indexes when evaluating queries which access complex objects. In our future work we will develop cost formulas for different storage structures and clustering strategies for being able to rank alternative query plans accordingly.

## 6. References

- Banc88 F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, F. Velez: *The Design and Implementation of O<sub>2</sub>, an Object-Oriented Database System*. K.R. Dittrich (Ed.), *Advances in Object-Oriented Database Systems*, Proc. 2nd Int. Workshop on Object-Oriented Database Systems, Bad Münster, Lecture Notes in Computer Science 334, Springer-Verlag, pp. 1-22, 1988.
- BeDe89 V. Benzaken, C. Delobel: *Dynamic Clustering Strategies in the O2 Object-Oriented Database System*. Altair, BP105, 78153 Le Chesnay Cedex, France, pp. 1-27, 1989.
- Dada86 P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, G. Walch: *A DBMS Prototype to Support Extended NF<sup>2</sup> Relations: An Integrated View on Flat Tables and Hierarchies*. ACM-SIGMOD, Proc. Int. Conf. on Management of Data Washington, D.C., pp. 356-367, 1986.
- DaLi89 P. Dadam, V. Linnemann: *Advanced Information Management (AIM): Advanced Database Technology for Integrated Applications*. IBM Systems Journal, Vol. 28, No. 4, pp. 661-681, 1989.
- DPS86 U. Deppisch, H.-B. Paul, H.-J. Schek: *A Storage System for Complex Objects*, K. Dittrich, U. Dayal (Eds.), Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, pp. 183 - 195, 1986.
- Gee77 W.C. McGee: *The information management system IMS/VS: Data base facilities*. IBM Systems Journal, Vol. 16, No. 2, pp. 96-123, 1977.
- HaOz88 A. Hafez, G. Ozsoyoglu: *Storage Structures for Nested Relations*. IEEE Data Engineering, Vol 11, No. 3, Special Issue on Nested Relations, pp. 31 - 38, 1988.
- HMMS87 T. Härder, K. Meyer-Wegener, B. Mitschang, A. Sikeler: *PRIMA - a DBMS Prototype Supporting Engineering Applications*. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, pp. 433 - 442, 1987.
- Ingr90 *INGRES/Database Administrator's Guide*. Release 6.3, 1990.
- KeDa91 U. Keßler, P. Dadam: *Auswertung komplexer Anfragen an hierarchisch strukturierte Objekte mit Pfadindexen (Evaluation of Complex Queries Against Hierarchically Structured Objects Using Path Indexes)*, H.-J. Appelrath (Ed), Proc. Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Springer-Verlag, Informatik-Fachberichte 270, pp. 218-237, 1991, (in German).
- KFC90 S. Khoshafian, M.J. Franklin, M.J. Carey: *Storage Management for Persistent Complex Objects*. Information Systems, Vol. 15, No. 3, pp. 303-320, 1990.
- Kim87 W. Kim, J. Banerjee, H.-T. Chou, J.F. Garza, D. Woelk: *Composite Object Support in an Object-Oriented Database System*. Proc. Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 118-125, 1987.
- Kim89 W. Kim, N. Ballou, H.-T. Chou, J.R. Garza, D. Woelk: *Features of the ORION Object-Oriented Database System*. W. Kim, F.H. Lochovsky (Eds.), *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Frontier Series, pp. 251-282, 1989.
- LLOW91 C. Lamb, G. Landis, J. Orenstein, D. Weinreb: *The ObjectStore Database System*. Communications of the ACM, Vol. 34, No. 10, Special Section: Next-Generation Database Systems, pp. 50-63, 1991.
- Lori85 R. Lorie, W. Kim, D. McNabb, W. Plouffe, A. Meier: *Supporting Complex Objects in a Relational System for Engineering Databases*. W. Kim, D.S. Reiner, D.S. Batory (Eds.), *Query Processing in Database Systems*, Topics in Information Systems, Springer-Verlag, pp. 145-155, 1985.

- Mits88 B. Mitschang: *The Molecule-Atom Data Model*. T. Härder (Ed.), The PRIMA Project Design and Implementation of a Non-Standard Database System, University Kaiserslautern, Report No. 26/88, Erwin-Schrödinger-Straße, 6750 Kaiserslautern, Germany, pp. 13-36, 1988.
- MSOP86 D. Maier, J. Stein, A. Otis, A. Purdy: *Development on an Object-Oriented DBMS*. Proc. Int. Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 472-482, 1986.
- Nier89 O. Nierstrasz: *A Survey of Object-Oriented Concepts*. W. Kim, F.H. Lochovsky (Eds.), Object-Oriented Concepts, Databases, and Applications, ACM Press, Frontier Series, pp. 3-21, 1989.
- Onto92 *ONTOS DB 2.2, First Time User's Guide*, 1992.
- Orac90 *Oracle RDBMS Database Administrator's Guide*, Version 6.0, 1990.
- Paul87 H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, U. Deppisch: *Architecture and Implementation of the Darmstadt Database Kernel System*. ACM-SIGMOD, Proc. Int. Conf. on Management of Data, San Francisco, USA, pp. 196-207, 1987.
- PiAn86 P. Pistor, F. Andersen: *Designing a Generalized NF2 Model with an SQL-Type Language Interface*. Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japan, pp. 278-285, 1986.
- RiSc93 C. Rich, M.H. Scholl: *Query Optimization in an OODBMS*. W. Stucky, A. Oberweis (eds.), Proc. Datenbanksysteme in Büro, Technik und Wissenschaft, GI-Fachtagung, Springer-Verlag, Informatik aktuell, pp. 266-284, 1993.
- Scho92 M. Scholl: *Physical Database Design for an Object-Oriented Database System*. J.-C. Freytag, G. Vossen, D.E. Maier (Eds.), Query Processing for Advanced Database Applications, Morgan Kaufmann, 1992.
- Scho92a M. H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch: *The COCOON Object Model*. ETH Zürich, Department of Computer Science, report no. 193, pp. 1-61, 1992. Also available at University of Ulm, Department of Computer Science, report no 93-02, pp. 1-59, 1993.
- ScPi82 H.-J. Schek, P. Pistor: Data Structures for an Integrated Data Base Management and Information Retrieval System. Proc. Int. Conf. on Very Large Data Bases, Mexico City, pp. 197-207, 1982.
- ScSc90 M. H. Scholl, H.-J. Schek: A relational object model. Proc. Int. Conf. on Database Theory (ICDT), Paris, France, Springer-Verlag, Lecture Notes in Computer Science 470, pp. 89-105, 1990.
- ScSi89 H. Schöning, A. Sikeler: *Cluster Mechanisms Supporting the Dynamic Construction of Complex Objects*. Proc. 3rd Int. Conf. on Foundations of Data Organization and Algorithms (FODO), Paris, France, Springer-Verlag, Lecture Notes in Computer Science 367, pp. 31-46, 1989.
- Sike88 A. Sikeler: *Key Concepts of the PRIMA Access System*. T. Härder (Ed.), The PRIMA Project Design and Implementation of a Non-Standard Database System, University Kaiserslautern, Report No. 26/88, Erwin-Schrödinger-Straße, 6750 Kaiserslautern, Germany, pp. 69-99, 1988.
- Stro86 B. Stroustrup: *The C++ Programming Language*, Addison Wesley, 1986
- TRSB93 W.B. Teeuw, C. Rich, M.H. Scholl, H.M. Blanken: *An Evaluation of Physical Disk I/O for Complex Object Processing*. Proc. 9th Int. Conf. on Data Engineering, Vienna, Austria, IEEE Computer Society Press, Los Alamitos, California, pp. 363-372, 1993.

## Appendix: Figures

```
/* Definition of a complex object. */
complex_object db_object_name [anchor_record_type = record_type_name] object_type [1]

/* Name of a complex object. */
db_object_name = string

/* User defined record type name of a record type. */
record_type_name = string

/* Recursively constructed complex object types. */
object_type = /* Examples of atomic value types. */ [2]
integer | real | var_string | fix_string(length) |

/* Definition of a set. */
set [implementation = implementation_type, [3]
     element_placement = placement_type] of object_type | [4]

/* Definition of a list. */
list [implementation = implementation_type, [5]
      element_placement = placement_type] of object_type | [6]

/* Definition of a tuple with a list of attributes. */
tuple (attribute_description {, attribute_description}*) [7]

/* Size of a fix length string. */
length = integer

/* Definition of an attribute of a tuple. */
attribute_description = attribute_name [location = location_type, [8]
                                       element_placement = placement_type]: object_type [9]

/* Name of an attribute. */
attribute_name = string

/* Parameters to define the storage structure of complex objects */

/* Definition whether an element of a set or list or an attribute of a tuple is stored inplace or in referenced records. */
placement_type = inplace | referenced (record_type_name) [10]

/* Alternatives to implement sets and lists. */
implementation_type = array | linked_list [11]

/* Definition whether the correspondent field of an attribute is localised within the primary block or a secondary block of the constructor data structure. */
location_type = primary | secondary (record_type_name) [12]
```

**Fig 8: Syntax to define the type and storage structure of complex objects**

```

complex_object Robots [anchor_record_type=Anchor_rec]
set [implementation=array, element_placement=referenced(Prim_rec)] of
tuple(
  R_nb [location=primary, element_placement=inplace]: integer, [1]

  Name [location=secondary(Sec_rec), element_placement=inplace]: fix_string(30), [2]

  Axes [location=secondary(Sec_rec), element_placement=inplace]: [3]
    list [implementation=array, element_placement=referenced (Axis_rec)] of [4]
      tuple ( [5]
        Axis_nb [location=primary, element_placement=inplace]: integer, [6]
        Matrices [location=primary, element_placement=inplace]: [7]
          list [implementation=array, element_placement=inplace] of [8]
            tuple ( [9]
              Row [location=primary, element_placement=inplace]: integer, [10]
              Vector [location=primary, element_placement=inplace]: [11]
                list [implementation=array,element_placement=inplace] of integer)), [12]

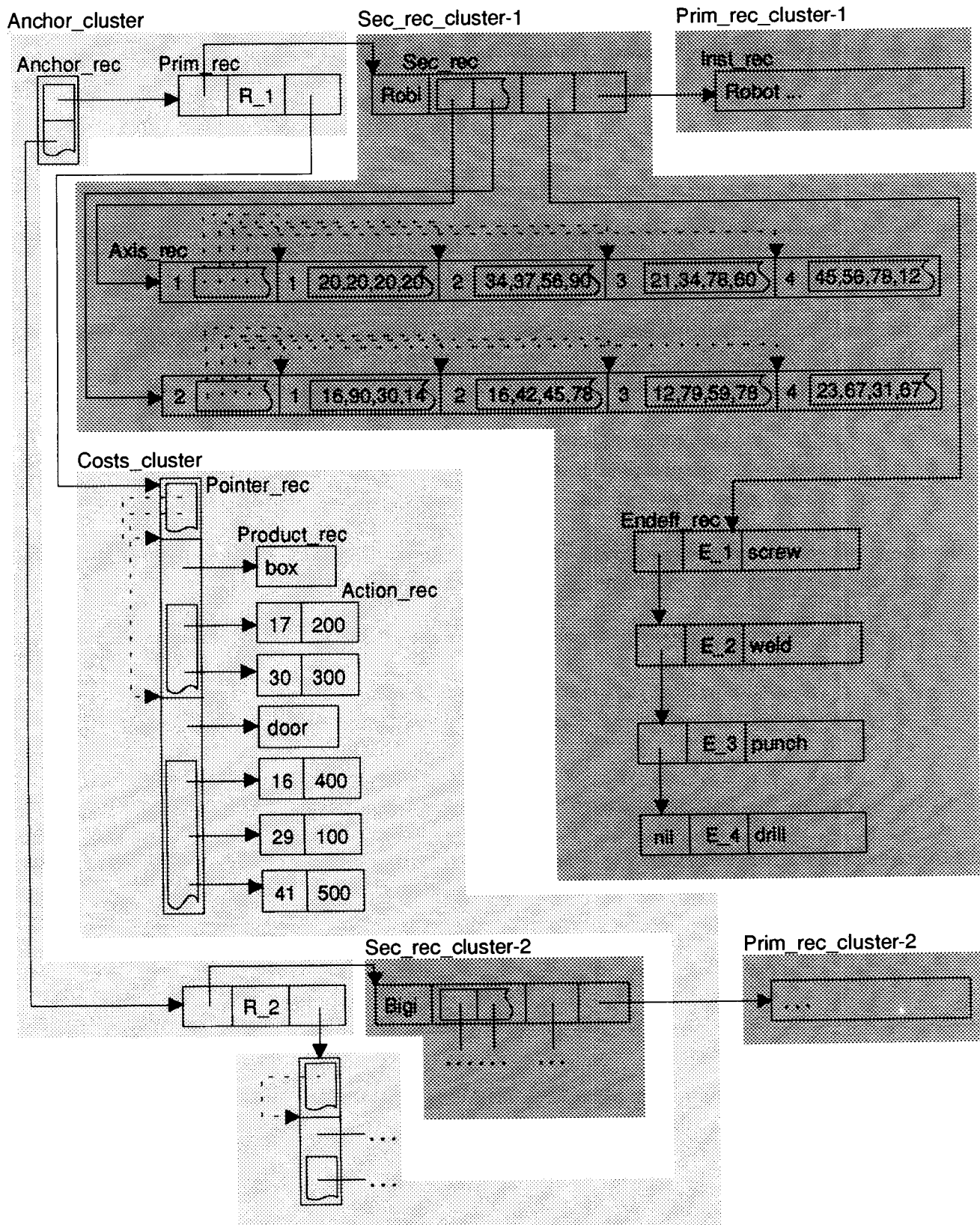
  Production_report [location=primary, element_placement=referenced (Pointer_rec)]: [13]
    set [implementation=array, element_placement=inplace] of
      tuple (
        Product [location=primary, element_placement=referenced (Product_rec)]: [14]
          fix_string(30),
        Action [location=primary, element_placement=inplace]:
          set [implementation=array, element_placement=referenced (Action_rec)] of [15]
            tuple (
              Week [location=primary, element_placement=inplace]: integer,
              Price [location=primary, element_placement=inplace]: integer)),

  Endeffectors [location=secondary(Sec_rec),element_placement=referenced(Endeff_Rec)]: [16]
    set [implementation=linked_list, element_placement=inplace] of [17]
      tuple (
        E_nb [location=primary, element_placement=inplace]: fix_string(30),
        Function [location=primary, element_placement=inplace]: fix_string(30)),

  Instructions [location=secondary(Sec_rec), element_placement=referenced (Inst_Rec)] [18]
    var_string))).

```

**Fig. 9: Type and storage structure definition of the eNF<sup>2</sup> Robots relation**



**object-centred record cluster** = 1 record cluster for each object  
**object-independent record cluster** = 1 record cluster for all records, independently of their object membership

Fig. 10: One feasible storage structure for the eNF<sup>2</sup> Robots relation

## Liste der bisher erschienenen Ulmer Informatik-Berichte

Einige davon sind per FTP von <ftp.informatik.uni-ulm.de> erhältlich

Die mit \* markierten Berichte sind vergriffen

## List of technical reports published by the University of Ulm

Some of them are available by FTP from <ftp.informatik.uni-ulm.de>

Reports marked with \* are out of print

- 91-01 KER-I KO, P. ORPONEN, U. SCHÖNING, O. WATANABE  
Instance Complexity
- 91-02\* K. GLADITZ, H. FASSBENDER, H. VOGLER  
Compiler-Based Implementation of Syntax-Directed Functional Programming
- 91-03 ALFONS GESER  
Relative Termination
- 91-04\* J. KÖBLER, U. SCHÖNING, J. TORAN  
Graph Isomorphism is low for PP
- 91-05 JOHANNES KÖBLER, THOMAS THIERAUF  
Complexity Restricted Advice Functions
- 91-06 UWE SCHÖNING  
Recent Highlights in Structural Complexity Theory
- 91-07 F. GREEN, J. KÖBLER, J. TORAN  
The Power of Middle Bit
- 91-08\* V. ARVIND, Y. HAN, L. HAMACHANDRA, J. KÖBLER, A. LOZANO,  
M. MUNDHENK, A. OGIWARA, U. SCHÖNING, R. SILVESTRI, T. THIERAUF  
Reductions for Sets of Low Information Content
- 92-01 VIKRAMAN ARVIND, JOHANNES KÖBLER, MARTIN MUNDHENK  
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets
- 92-02\* THOMAS NOLL, HEIKO VOGLER  
Top-down Parsing with Simultaneous Evaluation of Noncircular Attribute Grammars
- 92-03 FAKULTÄT FÜR INFORMATIK  
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen
- 92-04 V. ARVIND, J. KÖBLER, M. MUNDHENK  
Lowness and the Complexity of Sparse and Tally Descriptions

- 92-05 JOHANNES KÖBLER  
Locating P/poly Optimally in the Extended Low Hierarchy
- 92-06 ARMIN KÜHNEMANN, HEIKO VOGLER  
Synthesized and inherited functions -a new computational model for syntax-directed semantics
- 92-07 HEINZ FASSBENDER, HEIKO VOGLER  
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing
- 92-08 UWE SCHÖNING  
On Random Reductions from Sparse Sets to Tally Sets
- 92-09 HERMANN VON HASSELN, LAURA MARTIGNON  
Consistency in Stochastic Network
- 92-10 MICHAEL SCHMITT  
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent Any Linearly Separable Boolean Function
- 92-11 JOHANNES KÖBLER, SEINOSUKE TODA  
On the Power of Generalized MOD-Classes
- 92-12 V. ARVIND, J. KÖBLER, M. MUNDHENK  
Reliable Reductions, High Sets and Low Sets
- 92-13 ALFONS GESER  
On a monotonic semantic path ordering
- 92-14 JOOST ENGELFRIET, HEIKO VOGLER  
The Translation Power of Top-Down Tree-To-Graph Transducers
- 93-01 ALFRED LUPPER, KONRAD FROITZHEIM  
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications Manager
- 93-02 M.H. SCHOLL, C. LAASCH, C. RICH, H.-J. SCHEK, M. TRESCH  
The COCOON Object Model
- 93-03 THOMAS THIERAUF, SEINOSUKE TODA, OSAMU WATANABE  
On Sets Bounded Truth-Table Reducible to P-selective Sets
- 93-04 JIN-YI CAI, FREDERIC GREEN, THOMAS THIERAUF  
On the Correlation of Symmetric Functions
- 93-05 K.KUHN, M.REICHERT, M. NATHE, T. BEUTER, C. HEINLEIN, P. DADAM  
A Conceptual Approach to an Open Hospital Information System
- 93-06 KLAUS GASSNER  
Rechnerunterstützung für die konzeptuelle Modellierung
- 93-07 ULLRICH KESSLER, PETER DADAM  
Towards Customizable, Flexible Storage Structures for Complex Objects

# Ulmer Informatik-Berichte

ISSN 0939-5091

Herausgeber: Fakultät für Informatik

Universität Ulm, Oberer Eselsberg, D-89069 Ulm