Institut für Informatik der Universität Zürich

The KIDS Data Model Specification Language

Stefan Scherrer

Technical Report Nr. 93.14 April 93

The KIDS Data Model

Specification Language

Stefan Scherrer

Forschungsbereich Datenbanktechnologie
Institut für Informatik, Universität Zürich

Abstract

This preliminary report gives a guided tour of the syntax and semantics of the KIDS meta data model.

Table of Contents

1 Introduction and Motivation 2
2 Language Constructs 5
  2. 1 Overall Specification Structure 6
  2. 2 Identification 7
  2. 3 Interface 8
  2. 4 Realization 10
  2. 5 Reduction 13
  2. 6 Parameters 16
  2. 7 Properties 16
3 Meta Model Implementation 19
  3. 1 Generating Subsystem Interfaces 19
  3. 2 Generating Subsystem Implementations 19
4 An Example: The Specification of the NO^2 Data Model 32
  4. 1 Meta Value Sets 32
  4. 2 Meta Object Type 37
  4. 3 Meta Association Types 39
  4. 4 Summary 41
  4. 5 Subsystem 42
5 Conclusion 43
  5. 1 Related Work 43
  5. 2 Actual Status of Work and Future Work 46
6 Bibliography 47

December 4, 1992 12:55

1 Introduction and Motivation

Different database management systems (DBMS) offer different data structuring concepts addressing the requirements popping up in different application domains like order-processing systems, production-control systems, multimedia systems or design systems of any kind.

For quite a long time, the relational data model was considered to be universally applicable in all domains. However, the growing number of application domains with strongly diverging requirements to data models has beaten the consensus on the relational data model. Consequently a growing number of data models which offer different concepts entered the scene. Most of the systems implementing these models have been built up from scratch, thought the systems do not differ in a manner that would justify such a proceeding. One important reason for this phenomenon was the lack of an adequate DBMS construction methodology.

In the recent past, different methodologies have been proposed which all try to increase the reusability of work. The corresponding systems are often called extensible DBMS. They might be classified into kernel systems, toolkit systems, and generative systems. The former two are restricted on the reuse of implementation, i.e. the reuse of low-level program code, e.g. routines for B+tree management, buffer management, etc. Besides the reuse of program code, generative systems also facilitate the reuse of design which has previously been formalized using a high-level specification language. Current generative extensible database systems are rather limited with respect to the expressiveness of their specification language. Either they are limited on one aspect, or even worse, the supported aspect, e.g. data management is only poorly supported.

We claim the KIDS[1] methodology to DBMS construction [Dittrich 91] to overcome these weaknesses in that it currently covers three major DBMS aspects, i.e. data management, integrity management, and transaction management. In this report, we will concentrate on the data management aspect. We present a specification language which enables the specification of nearly arbitrary data models. Though the specification language allows the specification of a wide range of data models, we also claim that the necessary amount of time to realize such specifications is decreased by orders of magnitude. This holds especially, if the DBMS implementor (DBI) has access to a growing library of reusable data model specifications which is considered to be likely in the near future.

The KIDS approach to data model specification assumes that all data models offer concepts to define object types, value sets, and association types. An object type describes structural properties which are common to a set of objects. While objects can persist and populate extensions, values, in contrast, can persist only if being attached to an object. Value sets define values with similar structures. So, values may be used to characterize the state of objects. Associations serve to interrelate different objects. Association types are used to group associations interrelating objects of similar object types and with similar mathematical properties.

1. Kernel-machine based Implementation of Database Management Systems

The data model specification language therefore offers means to specify the possible value sets, object types, and association types. Since this specification language uses one abstraction layer above the usual value set or type terminology, we often refer to meta types when the specification of a group of possible concrete types is meant.

Specifying a type (no matter whether a concrete or meta one) usually involves the definition of the applicable operations to instances of that type. In case this is a meta type, two levels of abstractions have to be carefully distinguished. First, instances of a meta type (e.g. meta type entity type [Chen 76]) are concrete entity types (e.g. operations like create an entity type person). The functionality applicable to concrete types is part of the meta type specification as well as the functionality applicable to instances of concrete types (e.g. operations like delete the object with the print-name "Joe Cool"). The latter represents the second level of abstraction and is an obviously more "concrete" one than the former. As a first summary, the specification language allows both the specification of the DDL and DML operation signatures of both levels of abstractions.

Different levels of abstractions also play an important role when the operation signature are to be implemented. In our approach, the semantics of these operations is expressed in terms of operations on more simple structures which are formalized in the EAV-algebra [Geppert 92b]. Referring again to the levels of abstractions, the level of DDL and DML operations is realized using abstractions offered by the EAV algebra. Since the physical storage and retrieval of structures offered by the EAV algebra would be unpromisingly run-time inefficient, these simple structures are reduced to structures which are much more appropriate with respect to run-time behavior.

The following figure gives a summary of the different kinds of peoples entering the scene doing different things at different points of time.

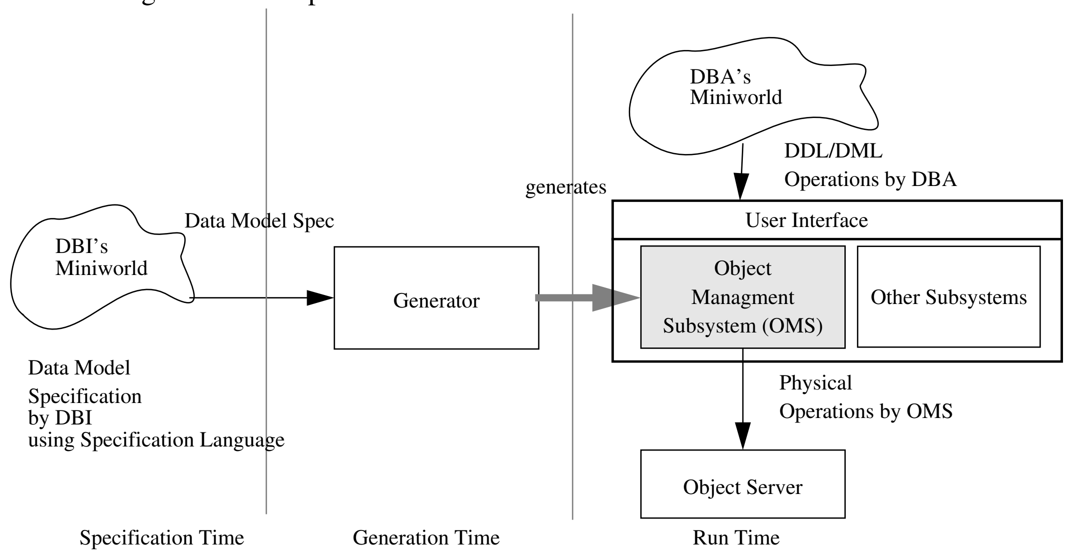

Figure 1 Different perceptions of DBMS with KIDS

The remainder of this report is outlined as follows. Section 2 introduces the constructs of the specification language in a top-down, syntax directed manner. However, examples appear sparsely in this section in order to keep it at finite length. Nevertheless, section 4 contains a major example; the specification of a structurally object-oriented data model using the concepts introduced in section 2. Section 3 discusses some implementation issues of the proposed specification language, i.e. gives first answers on questions arising in this context like how to (automatically!) derive object management subsystems from data model specifications. Section 5 concludes with a report on related work, our current status of work, and our future work schedule.

2 Language Constructs

In this section, the various constructs of the data model specification language are introduced and explained in a syntax directed manner where syntax fragments contain only the parts which are of actual interest. The design of those constructs was primarily driven by requirements listed in [Geppert 92a]. A major requirement has been the distinction of values, objects, and associations. Hence, different data models might are characterized according to their allowed value sets, object types, and association types with their characteristic operations, respectively.

The characterization has to be done using an abstraction layer above value sets, object types, and association types. For a particular data model to be realized using the overall KIDS approach [Dittrich 91], the allowed object types are therefore specified by some meta object types, the set of allowed value sets are described by the set of basic and complex meta value sets and, finally, the allowed associations between objects are constrained by some meta association types[1]. So, data model specifications are made up of each a set of meta object types, meta association types, and meta value sets (refer to grammar production[2] [1] to [4]).

```
dm-spec::= DM name                                                        [1]
              VALUE SETS instantiation-list                               [2]
              OBJECT TYPES instantiation-list                             [3]
              ASSOCIATION TYPES instantiation-list MD                     [4]
instantiation-list::= instantiation { "," instantiation }                 [5]
instantiation::= name "[" spec-actual-param { "," spec-actual-param } "]" [6]
spec-actual-param ::= instantiation | VALUE SETS | OBJECT TYPES |         [7]
              ASSOCIATION TYPES | ( "{" instantiation { "," instantiation } "}" )   [8]
```

The commonality of the different lines of production dm-spec is that each of them refers to [5] which describes how existing meta concept specifications are reused. The process of reusing a particular meta concept specification is called *instantiation*. In order to increase the reusability of meta concepts, a parameterization mechanism is introduced. In [6] is shown, how existing meta concepts are provided with actual parameters. The semantics of this parameterization mechanism will become clear as soon as the language constructs to specify new meta concepts are specified.

The following is an example data model specification which will be fully developed in section 4 though different pieces will occasionally appear in this section. It partially defines the concepts of the NO^2 structural object model [Dittrich 92]. Of course, many details are left open,

1. As a generalization of the terms *meta object type, meta value set,* and *meta association type*, the term *meta concept* is used.
2. Symbols in all capital denote keywords, whereas the others denote non-terminal symbols. "Key" characters are quoted. We use some kind of EBNF to specify the syntax. Curly brackets denote arbitrary repetitions, angle brackets denote options and slashes denote alternatives. Sometimes, different nonterminal symbols abbreviated by commas are used on the left side of productions for brevities sake.

DM NO^2	[9]
VALUE SETS	[10]
Integer, Real, LongField, Char, String,	[11]
Set[VALUE SETS], List[VALUE SETS],	[12]
Tuple[VALUE SETS], Array[VALUE SETS]	[13]
OBJECT TYPES	[14]
ObjectType[VALUE SETS]	[15]
ASSOCIATION TYPES	[16]
Ref[OBJECT TYPES]	[17]
Sub[OBJECT TYPES]	[18]
MD	[19]

since the specification instantiates many existing meta concept specifications (e.g. Integer, Set, Ref, etc.).

The specification indicates that the NO^2 type system knows some basic value sets (Integer, Real, LongField, Char, String) and some complex ones (Set, List, Tuple, Array). Note that the latter are instantiations of parameterized specifications. Furthermore, one kind of object types (ObjectType) is available and two kinds of association types (Sub, Ref) which are again specifications instantiated by providing actual parameters which are instantiations themselves.

As a short summary about the knowledge we have already acquired about the parameterization of specifications, we note that actual parameters are either instantiations or sets of instantiations. Furthermore note that predefined sets of specification instantiations (VALUE SETS, OBJECT TYPES, and ASSOCIATION TYPES) exist which collect the instantiations available in the current data model specification sections. Additional insights are given in 2. 6.

2. 1 Overall Specification Structure

Now, the language constructs to define new meta concepts either by reusing existing ones or by creating new ones are introduced in grammar productions [20] to [25].

Three means to create new meta concepts are distinguished [20]. The first one [21] corresponds to the specification of meta value sets, the second one [22] to the specification of meta object

new-spec::= mcvs-spec | mot-spec | mat-spec	[20]
mcvs-spec::= MCVS spec-formal-param-list body mcvs-properties SVCM.	[21]
mot-spec::=MOTspec-formal-param-list body mot-properties TOM	[22]
mat-spec::= MAT spec-formal-param-list body mat-properties TAM	[23]
spec-formal-param-list::= name “[“ spec-formal-param { “;” spec-formal-param } “]”	[24]
spec-formal-param::= “(“ name “:” INTENSION “;” name “:” EXTENSION “)”	[25]

types and the third one [23] facilitates the specification of meta association types.

Their similarity lies in their common structure. Each meta concept specification is named (e.g. ObjectType) and contains an arbitrary number of formal parameters [46], a body [26], and properties which are specific to each kind of meta concept (refer to subsection 2. 7). Before we explain the semantics of the formal parameters (refer to subsection 2. 6) let us consider the overall structure of a specification body.

The body of a specification itself is separated into an identification, an interface, a realization, and a reduction part ([26] to [29]).

body::=	identification	[26]
	interface	[27]
	realization	[28]
	reduction	[29]

While these different parts of specification bodies are further detailed in subsections 2. 2 to 2. 5, subsection 2. 6 defines the syntax and semantics of formal parameters of meta concept specifications and subsection 2. 7 defines the different properties that different kinds of meta concept specifications can be attributed with.

2. 2 Identification

In the identification part, means to *identify* the concepts offered by the data model of interest are defined. For both the concepts on the intensional (schema) level [30] and extensional (instance) level [32] concrete types may be defined whose instances allow to identify either the schema or instance level concepts.

identification::=	INTENSION IDENTIFICATION	[30]
	id-declaration	[31]
	EXTENSION IDENTIFIACTION	[32]
	id-declaration	[33]
id-declaration::=	identificator { "," identificator }	[34]
identificator::=	vs-name "(" name ")"	[35]
vs-name::=	SURROGATE | "String"	[36]

Concrete types defined in the identification part can be used lateron in the interface part to specify the signatures (see below). For an example, refer to [37] and [39]. The former defines the concrete types OTid and Name which simply represent a stock of values to uniquely identify the items on the intensional level while latter does the same on the extensional level. If we assume that this identification declaration were in meta object type specification which defines the syntax and semantics of NO^2 object types, a member of OTid would always reference an object type definition by means of a surrogate, whereas a member of Oid would always reference an object being instance of a particular object type. See section 4 in order to get the whole picture of the ObjectType specification.

```
INTENSION IDENTIFICATION                                    [37]
        SURROGATE(OTid), String(Name)                       [38]
EXTENSION IDENTIFICATION                                    [39]
        SURROGATE(Oid)                                      [40]
```

2. 3 Interface

In the interface part ([41] to [44]), all the relevant *operation signatures* which are lateron visible to the DBI for the implementation of the user interface. They define the functional syntax of each meta concept. Similar to the identification part, an intensional and an extensional level are distinguished. While the former refers to operations producing schema information, the latter deals with operations applicable to instances. Operation signatures can be defined according to productions [45] to [48].

```
interface:=     INTENSION INTERFACE                                                  [41]
                        { operation-signature }                                      [42]
                EXTENSION INTERFACE                                                  [43]
                        { operation-signature }                                      [44]
operation-signature::= name op-formal-parameter-list                                 [45]
op-formal-param-list::= "(" op-formal-param { ";" op-formal-param} ")" ":" concrete-type [46]
op-formal-param::= name { "," name } ":" concrete-type                               [47]
concrete-type::=name [ "(" concrete-type { "," concrete-type } ")" ]                 [48]
```

Each operation is named and can have an arbitrary number of formal parameters [45] and always returns an instance of a concrete type [46]. The interesting issue in this setting are the parameters and results of both the intensional and the extensional operations. Note that either intensions or extensions are both possible parameters in either of kinds of operations! For example to construct a particular value (e.g. a set-typed value), on one hand, it may be necessary to pass the corresponding element value set as actual parameter. On the other hand, it might be desirable to construct a value from other values directly. However, this fact is rather hard to represent in the syntax of the specification language. Hence, all we can say is that operations may have an arbitrary number of parameters and that the parameters and the result are typed using concrete types, which are in turn either sets of identificators (e.g. SetVSid) or type constructors already available in the data model specification (e.g. Set(String)).

The former is used to indicate that the DBI has no precise knowledge about the concrete element type. E.g. he might specify SetVSid if he just requires the resulting value to be a set value. On the other hand, if he knows exactly that actual parameters (or results) have to be sets of strings, he might declare the formal parameter type to be Set(String). One may question the story behind formal type declarations like Set(String), however they represent a means to attach more semantic information to operations specifications since the parameter is not only required to be a Set value but rather to be a Set of String value.

The following is an example interface specification of NO^2 object types together with the corresponding specification fragments we have encountered so far in the context of this example.

```
MOT ObjectType[ (VSid: INTENSION; Vid: EXTENSION) ]                                        [49]
IDENTIFICATION INTENSIONS                                                                  [50]
    SURROGATE(OTid), String(Name)                                                          [51]
IDENTIFICATION EXTENSIONS                                                                  [52]
    SURROGATE(Oid)                                                                         [53]
INTENSION INTERFACE                                                                        [54]
    ObjectType(vsid: VSid; nm: Name): OTid          /* create an object type */            [55]
    ObjectType(superOTid: OTid; nm: Name;           /* create an object type by inherit. */ [56]
                extendedAttNames: List(String); extendedAttVSids: List(VSid);              [57]
                refinedAttNames: List(String); refinedAttVSids: List(VSid)): OTid          [58]
    getVS(otid: OTid): VSid                         /* get vsid of object type */          [59]
EXTENSION INTERFACE                                                                        [60]
    create(vid: Vid; otid: OTid): Oid               /* create object given a value */      [61]
    create(otid: OTid): Oid                         /* create object with “empty” value */ [62]
    delete(obj: Oid): Boolean                       /* delete object */                    [63]
    putValue(oid: Oid; vid: Vid): Oid               /* replace old value by new value */   [64]
    getValue(oid: Oid): Vid                         /* get object value */                 [65]
    getOT(oid: Oid): OTid                           /* get object’s object type */         [66]
    identityp(oid1, oid2:Oid): Boolean              /* predicate to test for object identity */[67]
    equalityp(oid1, oid2:Oid): Boolean              /* predicate to test for object equality */
                                                                                           [68]
INTENSION REALIZATION                               /* to be introduced lateron... */      [69]
EXTENSION REALIZATION                                                                      [70]
INTENSION REDUCTION                                                                        [71]
EXTENSION REDUCTION                                                                        [72]
TOM.                                                                                       [73]
```

Lines [49] to [53] resume the examples on MOT ObjectType presented so far in this paper. In lines [54] to [59], the available operations on the intensional level are specified. Two object type constructors exist, both having equal name but different parameter lists. Moreover, when invoked, both result in an identifier for the object type intension definition, which can be primarily used lateron to create instances (see below). While the former constructor is invoked whenever an object type is defined from scratch the latter is used to refine and extend an existing object type.

Note the different parameter types of the operations, i.e. the VSid/Vid and OTid/Oid concrete types. The formers result from the formal parameter declaration of the MOT specification [49]. If the ObjectType specification is instantiated with { Set, List, Tuple, Array } as actual parameters, then VSid represents the union of the concrete types defined in the IDENTIFICATION INTENSION clauses of the respective specifications. Vid correspondingly reflects the union of concrete type introduced in the IDENTIFICATION INTENSION clauses. OTid, Name and Oid in turn are the newly introduced concrete types on then intensional level and extensional level ([50] to [52]), respectively. Whenever the specification ObjectType is itself used as an actual parameter of a meta concept instantiation (e.g. [17]), OTid and Name will be referenced by the keyword INTENSION and Oid iby the keyword EXTENSION in the formal parameter list of the parameterized specification.

In lines [60] to [68], the available operations on the extensional level are specified. Note that it is impossible to draw a sharp borderline between the operations appearing in the INTENSION and EXTENSION INTERFACE, since on both levels, intensional and extensional information is required for parameters and operation results. As a guide line, operations which are creating intensional information should be located on the intensional level whereas operations which are creating extensional information should be located on the extensional level. However, the location of accessor operations (e.g. [66]) is somehow arbitrary, since in the same operation both intensional and extensional information is either processed or resulting.

2. 4 Realization

We have presented above means how to express the syntax of a data model, i.e. the signatures of the operations. Now we show how the structural semantics of these operations is realized in a rather denotational approach in that the DBI provides a mapping of data model operations to EAV operations which are themselves formalized in the EAV algebra [Geppert 92b].

The structural semantics of each of the DML operations can be defined by describing its effects in terms of (hypothetically) creating or deleting EAV triples of a hypothetical triple database. In order to increase readability of the realizations, two sets of triples are distinguished. First, the set DB contains all triples which carry instance level information, and second, the set SCHEMA subsumes the schema information occurring in the context of the targeted data model. While operations on the intensional level typically operate on the latter set of triples, operations on the extensional level operate on the former.

From an object-oriented point of view, both databases are objects, whose state can be manipulated using different operations. Logically the state of such an object may be viewed as a set of triples (quadruples, resp.) which can be manipulated using the usual set operations like insert, delete, modify, image and selection. The selection operation is used to build arbitrary subsets of the triple database. The image operation is used to iterate over a subset of triples in order to apply an operation to each of the selected triples. In this context, operations are triple operations again or DML operations which have already been defined by the DBI. Three different kinds of results for functions exist:

- *Nothing at all*: Basic functions like triple create, delete and update, which merely affect the set of triples have side effects only.
- *A set of triples*: This is the typical result of the selection operation in order to subdivide the triple database into logical units.
- *A DML-structure*: DML operations always return DML structures usually identified by surrogates. Note that such a surrogate again characterizes a set of triples!

Indeed, only two kinds of results exist: nothing at all or sets of triples.

After having defined the abstract model to realize DML operations, the linguistic context has to be defined into which this abstract model is embedded. In particular, the question has to be answered whether[1] and which kinds of control flow concepts are needed to express the semantics of DML operations. The answer to the latter question finally leads to some kind of triple programming language which is briefly overviewed in the following.

The realization language offers a concept to define variables. Since the abstract model provides for sets of triples only, there is no need for typed variables. At any place variables may be defined. The value of a variable is defined set by a triple expression [87], e.g. a selection on the DB triple set. This offers the first declarative means to subsetting. The second means to describe a (potentially empty) subset is to denote a triple identifier. All triples matching this particular identifier are included into its subset can be referenced by the particular variable. The usual procedure to create a new subset of triples is to start with an empty set which is defined by the creation of a new unique triple identifier and may be extended using the typical insertion and deletion operations. However, note that the effects of the modification operations ([88] to [90]) take always place with respect to the universal set of triples. In other words, the variable concept allows to define views of the triple database.

After having defined the items programs can work on, we show the different ways the DBI can take influence on control flow. The first is to call an operation whereever a value is appropriate [83]. This may either happen whenever an actual parameter of an operation invocation has to be passed [83] or whenever something is assigned to a variable [82]. Note, that type compatibility is required in the former case. The next way to influence control flow is the IF statement [84] denoting the execution of either alternative depending on the value of a boolean expression [87] [2]. Additionally, there exists a WHILE statement [85], which allows iteration of a body while a boolean expression evaluates to true. Finally, the RETURN statement [86] allows to interrupt the execution of the operation and to pass an expression as the resulting value of the operation.

The following grammar rules syntactically summarize the concepts introduced in this subsection. The most important grammar rule is [74] to [77] which structures the meta concept realization into an intensional and extensional part.

1. We would be glad to follow a purely algebraic approach to semantics specification. Maybe, the current approach will reduce itself to an algebraic approach, as soon as the problem of semantics specification is better understood.
2. Maybe the syntax needs to be extended in order to capture logical connectors.

```
realisation::=    INTENSION REALIZATION                                      [74]
                        { operation-signature-implementation }               [75]
                  EXTENSION REALIZATION                                      [76]
                        { operation-signature-implementation }               [77]
operation-signature-implementation::= operation-signature block              [78]
block::= "(" { statement } ")"                                               [79]
statement::= assignment | if-statement | while-statement|                    [80]
                  return-statement | triple-operation-call                   [81]
assignment::= name ":=" selection | projection | value | NEW "(" name ")"    [82]
value ::= constant | name | name "(" [ value ] {"," value } ")"              [83]
if-statement::= IF boolean-expression THEN block ELSE block                  [84]
while-statement::= WHILE boolean-expression block                            [85]
return-statement::= RETURN expression                                        [86]
boolean-expression::= expression cop expression                              [87]
triple-operation-call::=  create-triple | create-quadruple |                 [88]
                          update-triple | update-quadruple |                 [89]
                          delete-triple | delete-quadruple                   [90]
create-triple::= "CT" eav-param-list                                         [91]
update-triple::= "UT" eav-param-list                                         [92]
delete-triple::= "DT" "ea-param-list ")"                                     [93]
create-quadruple::= "CQ" eav-param-list                                      [94]
update-quadruple::= "UQ" ea-param-list "," value "," index ")"               [95]
delete-quadruple::= "DQ" ea-param-list "," index ")"                         [96]
ea-param-list ::= "(" ( SCHEMA | DB ) "," entity "," attribute               [97]
eav-param-list::= ea-param-list "," value ")"                                [98]
entity, attribute::=name                                                     [99]
value::=value-expression                                                    [100]
index::=natural-number-expression                                           [101]
```

In the following ([102] to [115]), example realizations of each an intensional and an extensional operation of MOT ObjectType are illustrated. They correspond to signatures [55] and [62], respectively.

The former operation realization ([103] to [107]) represents the primary object type constructor given a value set identifier vsid and a name nm. Since new intensional information is stored, a means to identify it lateron is required. Hence, a new OTid is reserved [104] and assigned to the variable ot. The creation of an ObjectType has to be reflected in the triple database SCHEMA. Thus, two triples are created, both with entity ot. The former attaches the identifier of the value set using attribute vs [105] whereas the latter stores the object type name using the triple attribute name [106]. As the result of the operation, the newly create object type identifier is returned [107].

```
INTENSION REALIZATION                                                    [102]
    ObjectType(vsid: VSid; nm: Name) (                                   [103]
        ot:=NEW(OTid)                     /* get new identifier */       [104]
        CT(SCHEMA, ot, vs, vsid)          /* attach type info */         [105]
        CT(SCHEMA, ot, name, nm)          /* attach naming info */       [106]
        RETURN ot)                                                       [107]
EXTENSION REALIZATION                                                    [108]
    create(vid: Vid; otid: OTid) (                                       [109]
        IF otid.vs=vid.vs THEN (                                         [110]
            obj:=NEW(Oid)                 /* get new identifier */       [111]
            CT(DB, obj, ot, otid)         /* attach type info */         [112]
            CT(DB, obj, val, vid)         /* attach its value id*/       [113]
            RETURN obj)                   /* return oid */               [114]
        ELSE /* raise type mismatch error */ )                           [115]
```

The latter operation realization ([109] to [115]) illustrated here represents the primary object constructor given a value identifier vid and an object type identifier otid. The resulting object is required to be of object type otid and to have value vid attached with. Before creating a new object, the value vid is type-checked with the value set associated with object type otid [110][1]. On the successful type check, a new object is created by first creating a new object identifier which is assigned to obj [111]. As the next step, the object type otid and the value vid are attached to the newly created object obj by means of creating two triples with attributes ot [112] and val [113], respectively. Finally, the object identifier obj is returned [114]. On the unsuccessful type check a type mismatch error has to be signalled[2].

2. 5 Reduction

Since the straightforward realization of the abstract triple model using a triple machine currently is not possible due to efficiency reasons, the DBI provides a mappings between sets of triples and physical objects (*reduction rules*). The general objective of such a mappings is to group correlated information into a structure which can be accessed efficiently. The grouping in turn is expressed in terms of a triple selection.

The question that arises immediately is what are physical objects. One idea would be to directly map specific sets of triples to structures provided by the underlying object server [Nittel 92]. The problems with this idea are first that this interface is still not yet stable, second that the mapping seems to be a tedious task from a DBI point of view and third that the DBI is responsible for an optimal representation. We feel that this kind of task can be factorized out by not directly mapping triples to physical objects but to complex values (SET, LIST, TUPLE), whose

1. As an realization assumption, it is assumed, that the value set of every value is known by attribute vs.
2. No consensus on error management exists yet.

structures are again defined in the EAV-algebra. It is then up to the code generator (c.f. section 3. 2) to map the complex value accesses to physical object accesses. We feel that this approach offers many optimization opportunities. In other words, in order to facilitate reasonable efficiency of the object subsystem to be generated, the DBI *reduces* triple subsets to complex values by means of specifying reduction rules using the following language constructs.

```
reduction::=      INTENSION REDUCTION                                                      [116]
                        { reduction-rule }                                                 [117]
                  EXTENSION REDUCTION                                                      [118]
                        { reduction-rule }                                                 [119]
reduction-rule ::= declaration assignment selection                                        [120]
declaration ::= name “:” TUPLE “(“ PID “:” name { “;” variable-declaration } “)”           [121]
variable-declaration ::= name “:” ( name | SET “(“ name “)” | LIST “(“ name “)”            [122]
assignment ::= name “:=” “[“ PID “:=” expression { “;” variable-assignement } “]”          [123]
variable-assignment ::= name “:=” ( expression | complex expression )                      [124]
simple-expression ::= eav-ent | eav-val                                                    [125]
complex-expression ::= INSERT ”(“ name, simple-expression ”)” |                            [126]
                         DELETE “(“ name, simple-expression “)”                            [127]
selection::= “(σ[“ name “.” predicate “](“ (DB | SCHEMA) “)”                               [128]
predicate := value cop value | TRIPLE “(“ name “)” | QUADRUPLE “(“ name “)” |              [129]
                  predicate “∧” predicate | predicate “∨” predicate | “(“ predicate “)” |  [130]
                  “¬” predicate                                                            [131]
cop ::= “=” | “≠”                                                                          [132]
value::= eva-val | eav-att | TYPE “(“ eav-ent “)”                                          [133]
eav-ent::= ENT “(“ name “)”                                                                [134]
eva-att::= ATT “(“ name “)”                                                                [135]
eav-val::= name “.” name                                                                   [136]
```

According to [120], reduction rules can be separated into three major parts.

In the declaration part, the layout of the physical object is declared by means of a TUPLE complex value set declaration. Note that the first attribute name is required to be PID. This attribute will carry the physical object identifier lateron [121]. The attribute types must be either of basic value sets (e.g. String, Integer, etc.) or concrete types defined in IDENTIFICATION clauses (e.g. OTid, Oid) or complex values defined by the EAV algebra (e.g. LIST, SET) [122]. The information resulting implicated in such declaration helps to compute offsets within physical objects and gives hints how the physical data model of the object server (i.e. the references) may be used to find an optimal representation at subsystem-generation time.

In the assignment part, expressions where the current triple satisfying the selection predicate is involved into are assigned to the different components of the physical object. Hence, it may request several[1] triples to complete the information to create a physical object. This corresponds to the semantics of the term “reduction” which suggests the reduction of a set of triples to one physical object. The valid expressions assignable to physical components are given by [124]. In particular, the entity or value component of the current triple is allowed to become part of the physical object [125]. Moreover, if the component type of the physical object is either LIST or SET, manipulations of these structures using INSERT and DELETE are allowed as well [126]. Note that with respect to assignments, type compatibility is required and checked by the compiler.

In the selection part, a pattern is defined. Every triple matching the pattern can be referred to with a special variable in the afore-explained parts of reduction rules. The pattern is defined by means of logical predicates which may be defined according to rules [129] to [131]. As comparison operators, equality(=) and in-equality (≠) haven proven to be adequate up to now. From basic predicates involving the comparison operations, more complex ones can be built using the usual logical connectors. Rule [133] defines the ground symbols of the predicates. Since the predicates are evaluated at system-generation time, only information available at this point of time can be referred to, i.e. the type of the entity and the value component of the triple and the attribute name[1].

The following example ([137] to [144]) represents one pair of reduction rules to reduce the triples occurring in the MOT ObjectType. In line [138], the layout of the physical object is defined by means of defining a TUPLE complex value. The first attribute of the tuple is required to be the predefined PID attribute, whose value will represent the physical object identifier after reduction has taken place. In this example, the remaining attribute' values are required to be either of basic concrete types (e.g. String) or concrete types (e.g. OTid, VSid) declared in IDENTIFICATION section of the ObjectType specification. In line [139], the assignments of triple expressions to physical object components are defined. Expressions in turn refer to components (entity, attribute, value, index) of the triples/quadruples which match the pattern defined by the selection [140]. All the intensional triples (the SCHEMA database), whose entity is of type OTid and its attribute name is name or, again its entity is of type OTid its attribute name is vs satisfy the selection predicate. The current triple/quadruple matching the predicated is referred to as ot.

INTENSION REDUCTION	[137]
ObjectType : TUPLE(PID: OTid; name: String; vsid: VSid)	[138]
ObjectType:= [PID:= ENT(ot); name:=ot.name; vsid:=ot.vs]	[139]
(σ[ot. (TYPE(ENT(ot) = OTid) ∧ ((ATT(ot) = name) ∨ (ATT(ot) = vs))](SCHEMA)	[140]
EXTENSION REDUCTION	[141]
Object : TUPLE(PID: Oid; ott: OTid; value: Vid)	[142]
Object:= [PID:=ENT(o); ot:=o.ot; value:= o.val]	[143]
(σ[o. (TYPE(ENT(o) = Oid) ∧ ((ATT(o) = val) ∨ (ATT(o) = ot))](DB)	[144]

1. This corresponds to the semantics of the term "reduction" which suggests a reduction sets of triples to physical objects.
1. Note that the triple value component cannot be used since it is not available yet at generation time.

The reduction rule defines for each concrete ObjectType a physical object containing the name and the value set information. Similar, for each object which is instance of a concrete ObjectType, a reference to the to the intensional information and a reference to the value representing the state of the object are store in a physical object.

2. 6 Parameters

During the study of the previous subsection, the reader might have gathered different pieces of knowledge on the parameterization of specifications. This subsection's objective is to recapitulate this knowledge by means of measuring the concept at its requirements. The basic requirement that lead to the design of the parameterization mechanism was the ability to reuse specifications as much as possible[1]. The optimal concept would allow to pass (sets of) instantiations as actual parameters to generic specifications, and, within the generic specification, the parts of interest of the actual parameters to be referencable in a way that only meaningful parts are referencable. The only thing that we may confidently assume of specifications its that each of them opens a new subspace of identifiers for both intensions and extensions in the IDENTIFICATION section (refer to 2. 2). In other words, the only thing we can assume is that at least two subspaces are opened, one for intension identifiers and one for extension identifiers. We claim that our mechanism is optimal because each formal parameter allows to define *two* variables; one which references the subspace of intension identificators and one for the extension identificators. The variables are bound at instantiation time to the types introduced in the argument specification. These type variables may then be used within the generic specification to define its interface similar to other concrete types.

2. 7 Properties

Up to now, only generic language constructs like means to identify the entities their interfaces and their structural realizations have been considered, regardless of the differences of the different kinds of meta concepts namely MCVS, MOT and MAT. In this subsection, we introduce their characteristics and defined language constructs which allow to describe these characteristic in particular meta concept specifications. The separation of the language into "kind-specific" language constructs and "kind-independent" constructs makes it easy to track (likely) future changes to the language, i.e. since we feel that the "kind-specific" construct are not mature yet.

While we have not yet found any specific properties for MCVS, the next two subsections describe the properties of MOT and MAT.

2. 7. 1 Meta Object Types

According to the requirements specification, the DBI has to specify how objects obtain persis-

1. We are very well aware of that parameterization is just one means to address this requirements and not sufficient. For example, a query language to retrieve the specifications of interest are necessary as well. However, by following the fundamental principle "one problem at time", we concentrate on parameterization here.

tence and how they populate (possibly user-defined) extensions.

The extension-options currently considered to be useful are to define whether the DBA is allowed to define own extensions to collect instances of concrete object types or whether the system defines one whenever a concrete object type is created, or whether there exists no extensions at all.

In order to specify the persistence property of a meta object type, the DBI has to define whether and how the DBA will lateron be able to declare objects to be persistent and what the default behavior of the system is when objects are created. The DBA defined persistence options are to define objects being persistent as soon as they are contained in a particular extension or on an object-by-object base.

```
mot-properties::=                                                         [145]
            PERSISTENCE persistence                                       [146]
            EXTENSIONS extension                                          [147]
persistence::= [user-defined-persistency] default-persistency             [148]
user-defined-persistency::=USER-DEFINED ( PER-EXTENSION | PER-OBJECT )    [149]
default-persistency::= DEFAULT-PERSISTENT | DEFAULT-NON-PERSISTENT        [150]
extension::= ONE-PER-TYPE | USER-DEFINED | NONE                           [151]
```

The hints given in the persistence and extensions clause allow the generator to construct interface routines and additional code e.g. for extension management. Refer to MOT ObjectType in section 4 as an example.

2. 7. 2 Meta Association Types

Our perception of associations is strongly similar to the mathematical notion of relations. So if the DBI introduces a new association concept by specifying a meta association type, he has first to specify the allowed arity of the association types to be specified by the DBA lateron, second the entities allowed to be involved in associations. Moreover, since associations correspond to mathematical relations, their mathematical properties are of interest as well.

Besides the features common to all aforementioned meta concepts, the DBI has to specify propagation semantics for each meta association type. If a particular operation is applied to one participated entity, the effects on the other participated entities can be given in terms of propagation clauses which are in fact some kind of trigger definitions.

A propagation clause [154] covers the definition an event and an action. The former is defined by using the keywords BEFORE and AFTER in relationship with some kind of operation signature. In contrast to usual operation signatures, the name and the number of formal parameters have to match with an existing operation signature. Potential ambiguities can be resolved by denominating the operation using the meta concept name. Note that the formal parameters are not typed since this would be redundant with respect to the existing specifications. As usual, the formal parameters can be used within the block to express the action semantics of the propagation clause. Note that therefore the same language constructs are available as for the specification of operation realizations. Logically[1], if the running system issues an operation for which an according BEFORE or AFTER propagation clause exists, the appropriate block has to be executed.

Having this kind of perception of association types, inheritance is not considered to be modeled using a meta association type since we mainly view it as a mechanism to derive new type definitions from existing ones. So the place to specify that is clearly in the INTENSION INTERFACE of the MOT specification. The semantics may then be expressed by implementing the constructor in a specific way.

The specification language introduces the following grammar productions to allow the definition of the artifacts enumerated above:

```
mat-properties::=                                                              [152]
                    PROPERTIES                                                 [153]
                         properties                                            [154]
                    PROPAGATION                                                [155]
                         { trigger-declaration }                               [156]
properties::=       (RELATIONAL [TRANSITIVE] [REFLEXIVE] [SYMETRIC]) |         [157]
                    (FUNCTIONAL [INJECTIVE] [SURJECTIVE] [BIJECTIVE])          [158]
trigger-declaration::= (BEFORE | AFTER) event-specification block              [159]
event-specification::= name "." name "(" name { "," name } ")"                 [160]
```

Refer to section 4 for some comprehensive examples of MAT specifications!

1. Of course such an event detection and action execution has to be very efficient. Since the triggers are defined at DBMS specification time, we believe that many statical optimizations are possible.

3 Meta Model Implementation

The implementation of the meta model presented above can be separated into four major subtasks:

- A model to represent the concepts of the data model at the subsystem interface has to be designed. This includes answers to questions like: What is the result of the invocation of a value set constructor? What is the result of the invocation of a value constructor?
- A programming language for OMS implementation has to be evaluated. A related issue is whether the subsystem interface should be offered in different languages. We prefer to concentrate on one language that is powerful enough to ease the third and fourth item and that can be called from other languages very well.
- An algorithm has to be developed to automatically derive the OMS interface from data model specifications, i.e. from their interface part. The derived interfaces must conform to the model resulting from the first task. From a DBI point of view, this generated interface serves as basis (together with traditional compiler construction tools, e.g. lex, yacc, etc. or graphical user interface builders, e.g. devguide) to implement query languages or graphical user interfaces.
- An algorithm has to be developed to automatically derive the implementation of the functionality resulting of the previous task.

In this report, we mainly concentrate on the last subtask due to space limitations.

3. 1 Generating Subsystem Interfaces

The general idea is to generate a programmer's library interface which obeys the following constraints. First, the interface must represent the data model concepts in the approach selected resulting from the first subtask. Second, the interface itself has to be expressed in terms of the programming language evaluated in the second subtask. Since the development of an algorithm which derives the subsystem interfaces is considered to be rather straightforward, we will concentrate on principles of generation of interface implementations due to space limitations.

3. 2 Generating Subsystem Implementations

The generator algorithm to be developed takes the specification of the signatures, their triple[1] implementations and the reduction rules as input and produces a set of operation implementations where each of them invokes operations on physical data structures which obey the data model provided by the underlying object server [Nittel 92].

In the following, assume a set of reduction rules and a set of DML operation realizations to be given. However, before investigating the principal mapping of triple expressions occurring in signature realizations to physical object operations, note two important constraints which have

1. In this subsection, we usually employ the term *triples* to implicitly denote both triples and quadruples.

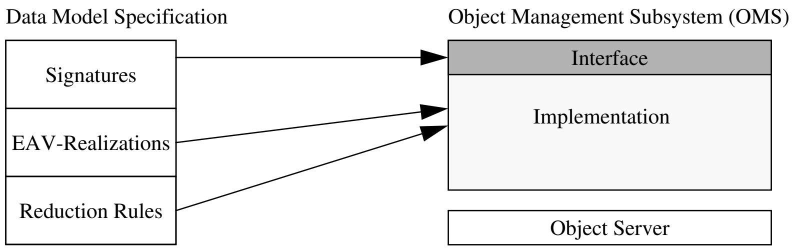

Figure 2 Overall approach

to be respected by the input of this algorithm:

- *completeness constraint*: Every triple that is hypothetically created/modified/deleted within an operation realization must be reduced by *at least one* reduction rule. *Exactly one* would be a too stringent constraint since we allow the redundant storage of information.
- *correctness constraint*: When retrieving objects using triple operations, no contradicting information must result. This is important in the context of redundantly stored information, i.e. if the "same" triple is reduced by two reduction rules. When modifying or creating an object, all the reduction rules have to be considered in order to enforce consistency of redundant information.

Since reduction rules declaratively describe the mapping of a set of triples to a set of complex values, we may perceive them as assertions between two sets which have to be satisfied at any time. Thus, if we change the left-hand side, i.e. change the set of triples and quadruples, e.g. if triples are created, deleted or manipulated, we have to take some corresponding actions at the right-hand side in order to keep the assertion satisfied. Furthermore, the complex-value action on the right-hand side has to be transformed to an action on the physical object.

Figure 3 gives an overview of the different operations occurring on the left-hand side. The operations CT, CQ, DT, DQ affect the set of hypothetical triples/quadruples by explicitly inserting or deleting elements to it. The operations UT, UQ and triple/quadruple selection and projection operate not on the set as a whole but on single elements. A mixture of both settings is the triple image operation which operates on a set of triples and applies an operation to each element of

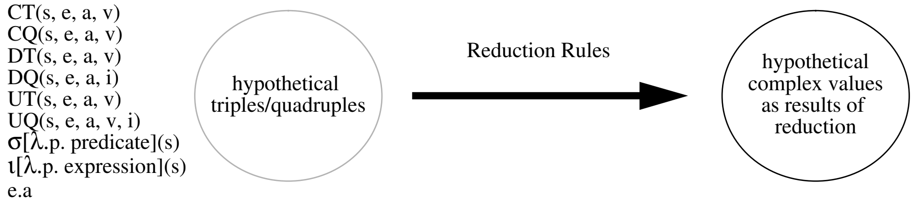

Figure 3 Generator model

this set.

The next lines resume the general structure of a reduction rule.

```
v: ComplexValueSet                                            [1]
v:= ComplexValueCreation(Triple-Selection)                    [2]
```

In the line [1], a complex value is declared. Currently, SET, TUPLE and LIST are available to orthogonally declare complex values. Line [2] describes how the complex value is constructed. Triple-Selection defines the subset of all hypothetical triples to be mapped to complex value v by the actual reduction rule. In fact, the selection predicates denotes a pattern for a set of triples. Since we allow redundant storage of information, the different subsets need not necessarily to be pair-wise disjunctive.

Whenever the set of triples is modified by a triple modification operation, a pattern matching happens. If the processed triple parameters satisfy the selection predicate, a match is detected and, depending on the action taken on the triple, code is attempted to be generated according to the ComplexValueCreation. Note that different triple actions may be necessary in order to complete the gather of information needed in order to generate code.

Two example reduction rules are shown below. They will reappear in section 4. While the first one describes the mapping of schema information, the second one concerns the mapping of instance information.

```
INTENSION REDUCTION                                                          [3]
    SetValueSet: TUPLE(PID: SetVSid; evs: VSid; name: String)                [4]
    SetValueSet:= [PID:=svs; evs:=svs.elem-vs; name:=svs.name]               [5]
         (σ[svs.  (TYPE(ENT(svs) = SetVSid) ∧                                [6]
                        ((ATT(svs) = name) ∨ (ATT(svs)=vs))](SCHEMA))        [7]
EXTENSION REDUCTION                                                          [8]
    SetValue: TUPLE(PID: SetVid; vs: VSid; elements: SET(Vid))               [9]
    SetValue:= [PID:=sv; vs:=sv.vs; elements:=INSERT(elements,sv.elem)]      [10]
         (σ[sv. (TYPE(ENT(sv) = SetVid) ∧ ((ATT(sv) = elem) ∨ (ATT(sv) = vs))](DB))   [11]
```

Line [4] specifies the layout of the complex value to store value set information. In this example, the choice was made to store DML set-value-set schema information in tuples. Note that in this TUPLE declaration, only the complex value set constructors (SET, TUPLE, LIST) and basic value sets including the different identification value sets (e.g. VSid, SetVSid, SetVid) are allowed to appear[1]. Line [5] first defines how the complex value is actually constructed given that the

1. Note that the value for storing the schema information is flat whereas the value for storing the set value is a TUPLE whose third attribute is a SET [9].

necessary information is complete. Second, it defines patterns of the triples which are assumed to deliver the required information to construct a complex value [6]. The information contained in a triple (e.g. a formal parameter name appearing in the DML operation signature) is then assigned to the reduction rule. In this context, this kind of attachment is called *instantiating a reduction rule*. A reduction rule is *fully* instantiated, if its assignments are all attached.

Note, that it may need several triple operation invocations in order to fully instantiate a reduction rule action. Hence, all the triple creations within an DML operation realization are grouped in a sense that they collectively try to complete an reduction rule instantiation. The compiler opens a new *scope* with active reduction rules for each DML operation to be compiled. A reduction rule being visible in such a scope is referred to as *active* reduction rule. Each new kind of triple operation appearing in the program text leads to a new active reduction rule. If the DBI attempts to activate the same reduction rule more than once with the same kind of operation, and there is no way to attach the information carried by the triple operation to the reduction rule, the compiler aborts with an error message[1].

In case of the compiler reaches the end of a program block and active, not yet completely instantiated reduction rules exist in the corresponding scope, an error will be signalled that code generation was not possible since the net effect of the triple actions does not make sense with respect to the given set of reduction rules. In the other case, code generation may start now, according to the complex value expression of the reduction rule.

In the following, for each operation applicable to triples, an algorithm is sketched which describes how each kind of triple operation may participate in the generation of one physical object access. This participation has to happen in a way that the assertion semantics of the affected reduction rules is preserved. In a first step, the mapping of the triple changes to changes of the corresponding complex value is algorithmized (3. 2. 1). The issues with respect to control flow are then discussed (3. 2. 2). The mapping of the complex value operations to physical object operations represents the third step (3. 2. 3). Finally, the algorithms to manipulate reduction rules are outlined (3. 2. 4).

3. 2. 1 Basic Triple Operations

Creations

The following algorithm is applied whenever the parser encounters a CT statement having arguments s, e, a, and v passed by. Note that the parser has already collected/derived information like the type of s, e, and v (e.g. SetVid) either from the DML operation's parameter list or of statements similar to s:=NEW(SetVSid). This additional semantic information helps to organize the search for the affected reduction rules in question. The search itself is outlined in the match algorithm shown in section 3. 2. 4.

1. This scoping scheme will lead to a programmer's guide in order to obtain compilable programs. Note that compilation may abort not only due to syntactical program defects but also due to semantically unreasonable utilization of the programming constructs. In fact, experiences will lead to a refinement of the syntax in order to avoid the latter syndrome.

```
prepareCT(s, e, a, v) (                                              [12]
        FOR EACH rr IN match(s,e,a) (                                [13]
                include(instantiate(rr,e,a,v), ctrr))                [14]
```

Line [13] above indicates an iteration over the relevant reduction rules. This is necessary since several reduction rules applicable to one triple may potentially exist. In line [17], the call to instantiate(rr, e, a, v) attempts to substitute as many parameters as possible of the reduction rule action. If the parameter is already substituted, the matching reduction rule is copied and provided with the new actual parameter, and the actually instantiated reduction rule is returned as the result of the call. The result of this call is then passed to the include function which appends it to the current set of creation-reduction-rules (ctrr) unless it already is an element of it. This closely corresponds to the aforementioned scoping scheme.

Deletions

While the code generation preparations for the creation of quadruples is similar to the creation of triples, it is some more interesting to discuss the procedure for triple/quadruple deletions. For the deletion operations appearing in a operation signature realization, a new scope with active reduction rules is opened. The instantiation is similar to above.

```
prepareDT(s, e, a, v) (                                              [15]
        FOR EACH rr IN match(s,e,a) (                                [16]
                instantiate(rr, e, a, v)                             [17]
                include(rr, dtrr))   /* includes rr in the deletion reduction rules */   [18]
```

The difference between deletion and creation shows up whenever the compiler encounters the end of a DML operation realization. While code for the complex value expression is generated in the triple creation case, code for the inverse of this expression is generated in the deletion case. Note that the need for an inverse of the complex value expression imposes a restriction to the kinds of expressions that are allowed.

Projections

Given a triple projection operation, e.g. e.a, the task is to generate code which identifies the corresponding portion in the physical object. The information packed in the reduction rules can be used to obtain the solution.

```
generate-projection(s, e, a)(                                        [19]
        generate-call(generate-extraction(find-any-rr(s, a), e)      [20]
```

The first step is to find a matching reduction rule. The function find-any-rr uses the match algorithm in order to find all reduction rules in question, and then selects the one promising the most easiest access to the desired information[1]. The resulting reduction rule is then passed to generate-extraction which produces a function with one formal parameter, the physical object-identifier, only. The type information of the reduction rule allows to compute the offset within the physical object referenced by the formal parameter of the function. However, this simple scheme requires the components of the physical object to be of statical size[2].

An interesting problem is the code-generation for path expressions, e.g. an statement like:

```
t:=e.a1.a2                                                        [21]
```

The first component of the path (e.a1) denotes a projection in the aforementioned sense. Another projection (a2) is applied on the result of this projection. The compiler first has to generate a call to the first projection whose result is stored in an intermediate variable at runtime. Then a call to the second projection is generated with the variable as actual parameter. The algorithm can be recursively defined, assuming that the path is given by a list.

```
generate-projection(s, e, path)(                                      [22]
    IF length(path) = 1 THEN (                                        [23]
        generate-call(generate-extraction(find-any-rr(s, a), e))      [24]
    ELSE (                                                            [25]
        v:= create-new-variable-name()                                [26]
        generate-assignment(v, generate-call(                         [27]
                    generate-extraction(find-any-rr(s, car(path)), e)) [28]
        generate-projection(s, v, cdr(path))))                        [29]
```

Updates

At a first glance, two solutions to the code-generation problem of triple/quadruple updates seem to exist. First, a triple update might be resolved by first deleting it and the creating a new one. Second, code could be generated to identify all[1] the places where the information is stored and then to replace the data. The latter approach is preferred since the former always result in the creation and deletion of physical objects due to the reduction rule semantics what is not considered to be very efficient. Furthermore, the necessary information to create the modified physical object would be missing anyway.

---

1. Remember, we follow a read one write all policy to manage redundant information.
2. A workaround could be to store a table with the attribute number as index and the corresponding offsets as values whenever physical object components with variable size exist (see below).

```
generate-update(s, e, a, v)(                                              [30]
    rr:=match(s, e, a)                                                    [31]
    FOR EACH r IN rr (                                                    [32]
        o:= create-new-variable-nam()                                     [33]
        generate-assignement(o, generate-call(generate-extraction(r, a), e))   [34]
        generate-replace(e, o, v)                                         [35]
```

As usual, the operation match determines all relevant reduction rules whose predicate is satisfied by the triple of concern [31]. For each of these reduction rules, code is generated to determine the offset where component a is located under the assumption that e references a physical object identifier [33]. The offset is stored in a variable whose name is stored in o. Having both, identifier and offset, code is generated to replace the old piece of data by the new one represented by v. The corresponding algorithm is illustrated in 3. 2. 3.

Selections / Images

The next operation to be analyzed is the triple selection. Since selection has no effects to the database, no necessity for the generation of code to keep the set of reduction rules consistent exists. Selection expressions only appear in the context of triple image or in assignments. In other words, selection expressions are mostly used to qualify image operations more precisely. Hence, the information can be used to both optimize the code generation process (in that fewer reduction rules might be analyzed) and the generated code (in that more information to retrieve the desired information exists). When executed, the result of generated selection code is to provide a set of physical identifiers, where the image functions will be applied on each of them.

3. 2. 2 Flow of Control

Different kinds control statements have been introduced in section 2. In this subsection, their translation to executable code is illustrated. The general approach to induce over the structure of program texts. The anchor of the induction has already been illustrated in the last few subsections on the triple manipulation operations. Now, the influences of the programming language constructs to code generation are considered.

DML Operation Calls

The most decent way (at least from a DBI point of view) to express the semantics of operations is to express it in terms of already defined (and successfully compiled) DML operations. Consequently, we allow arbitrary calls from an operation realization to already defined operations.

For the transformation of operation calls, two basic alternatives exist. Either the triple realization corresponding to the called operation is included (with appropriate parameter substitution) and further processed (macro semantics) or a function call is inserted into the code such that the function is called at runtime (functional semantics). While the latter provides more flexibility and more compact code, more optimization opportunities seem to exist for the former. Since the issues for the choice have not yet been completely elaborated on, the choice still remains open. However, the important point is that code generation is possible in either case.

DML operation calls can be nested, i.e. one may directly pass the result of one function to another by passing a call to the former to the latter as actual parameter. Hence, a preceding rule is required which specifies the order in which the actual parameters are evaluated. In our context, the source code is rewritten in that the evaluation of the parameters is stacked from right to left where the operations in the stack are executed top-down in a sequential manner. The issues arising with sequences are described next.

Sequences

One of the most simple programming language constructs for flow control is the sequencing of statements (e.g. triple statements, operation calls, assignments, and control statements). The compiler tries to group calls to the triple database in order to generate code for groups of calls. In this section, we show the basics of this grouping mechanism.

In order to illustrate the main problem, assume the following (pathological) source code fragment which could be part of an data model operation realization:

```
CT(DB, e1, a1, v1)                                                   [36]
h:= e1.a1            /* yields v1 */                                 [37]
CT(DB, e2, a2, h)    /* trying to create triple (e2, a2, v1)         [38]
```

The triple projection in [37] assumes that triple (e1, a1, v1) is already in the database although the creation of triple (e2, a2, v1) might have been necessary to complete the instantiation of a reduction rule in order to create a physical object. Hence, insertion-code is generated at the place of line [38]. Nevertheless, code has to be generated to compute h.

The conclusion the reader should be able to draw from this simple example is that dependencies between the different (kinds of) statement of the program text to be compiled exist. The compiler has to detect harmful dependencies and either to abort code generation or to take them into account, i.e. when generating code for alternatives and iterations. The generation of code for the sequential execution of basic triple operations like CT and DT does not offer any problems. However, their interrelationships to the other operations have to be carefully described.

Projection & Selection

The primary objective of these two operations is the retrieval of information from the hypothetical triple database. An illustration of the problems has already been given in the previous subsection. Here, we describe how the compiler has to circumvent them.

When the compiler detects a projection or selection operation, he has to check whether semi-instantiated reduction rules already exist which match the projection (or selection predicate). If so, this means that the compiler has previously processed CT or DT operations but still has not yet enough information in order to generate corresponding code. If this information does not come up at the end of the DML-operation, code generation for the CTs and DTs is not possible anyway. If the necessary information is provided after the projection (or selection) operation, code generation is again not possible since semantically incorrect code would be produced.

As a summary, the only way to produce code for projection and selections is when they do not match any previously semi-instantiated reduction rule. Note that fully-instantiated reduction rules do not harm since their code is determinable before encountering the projection (selection, reps.). The knowledge on being fully-or semi-instantiated is manifested in the find-any-rr algorithm.

Alternatives (IF-statement)

The IF-statement splits the flow of control into two branches where one of them is going to be selected at runtime according to the result of a condition evaluation. In the end, both flows merge into the flow starting at the successor statement of the IF-statement. Since the evaluation of the condition may involve retrievals from the database, similar problems arise as illustrated in the previous subsection.

However, the situation is sightly more complicated in that the compiler has to generate code for both of the alternatives. Thus, first, the compiler processes the condition C. A successful processing of C allows to rewrite the source program as shown below, where X and Y stand for the entire statement sequence before resp. after the IF-statement, A and B for the first resp. second alternative of flow of control. Then the compiler continues (or restarts) to generate code for both of the alternatives of the rewritten program.

```
X                                                              [39]
IF C THEN (                  if(call(code-for-c)               [40]
    A)          --->              call(code-for-XAY)           [41]
ELSE (                       else                              [42]
    B)                            call(cod-for-XBY)            [43]
Y                                                              [44]
```

At a first glance, the involvement of Y in code generation of the IF-statement may look strange; but note that first the semantics is preserved by the code rewriting above and second that Y may contain statements which provide information needed to generate code for either of the alternatives. The reader may verify that program rewriting is possible for nested alternatives, too.

Iterations (WHILE-statement)

Below, the general situation of an iteration in the context of a program texts control flow is given.

```
X
WHILE C (A)
Y
```

The treatment of `C` follows the approaches illustrated above. If `C` refers to items involved in semi-instantiated reduction rules of source code fragment `X`, the compiler abort code generation immediately. In a next step, each operation of `A` is examined with respect to a possible contribution to a semi-instantiated reduction rule. Due to the sequential semantics of `A` itself, the operations of `A` are restricted either:

- to contribute to one single reduction rule, resulting in that the code implementing `A` and the iteration itself is attached just behind the code implementing the reduction rule action.
- to contain DML-operations only, resulting in that code can be immediately generated.

So, the DBI is rather restricted concerning the utilization of the iteration. This may indicate that opportunities to improve the actual programming constructs still exist.

3. 2. 3 Code Generation

In this section, the remaining algorithms are outlined which have been utilized in the previous subsections. Some algorithms are described rather informally because they both rely on the object server functionality and are still subject of changes.

The algorithm `finalize` shows the way the parser proceeds when encountering the end of a DML operation realization.

```
finalize() (            /* assumes active reduction rules to be globally declared.*/
            FOR EACH ct IN ctrr (
                        IF fully-instantiatedp(ct) THEN (
                                    generate-create-po(ct))
                        ELSE
                                    generate-update-po(ct)
            FOR EACH dt IN dtrr (
                        IF fully-instantiatedp(dt) THEN(
                                    generate-delete-po(dt))
                        ELSE
                                    generate-update-po(dt))
```

It is important to note that the code resulting of `generate-create-po` and `generate-delete` is not appended at the end of the already existing code for the operation being subject of compilation, but has rather to be placed where the corresponding triple operations appeared in the program structure. So the corresponding operations have to be grouped and marked with the corresponding reduction rule identifier at parsing time in order to fixate the generated code.

In the following, we show the different generation algorithms used in the previous subsections.

```
generate-create-po(rr: fully-instantiated-reduction-rule)(                    [59]
        s:=calculate-object-size(rr)                                          [60]
        generate(rr.PID:=create-physical-object(s) /* this is to start with only */)  [61]
        FOR EACH attribute IN rr                                              [62]
                generate-map(attribute, rr.PID, index)))                      [63]
```

The algorithm above indicates how the compiler deals with fully instantiated reduction rules (whose instantiation was triggered by CT operations) when encountering the end of an operation realization. First, the type information contained in the reduction rule is used to compute the initial size of the physical object. A call creating the physical object is generated next. The object identifier (which will result if the call is executed at runtime) is bound to the predefined attribute PID of the reduction rule in order to make the connection between physical and logical (triple) objects. Then, code is generated to map all the assignements of the reduction rules to the physical object [63]. Generate-map describes how assignments occurring in reduction rules are used to create in incremental object layout. The first and second parameter denote the assignement and its position in the reduction rule while the third parameter represents a variable name which will reference the physical object identifier at run time.

```
generate-map(attribute, index, poid-var-name)(                                [64]
        - create the next offset table entry: offsets[index+1]:=offsets[index]+size(attribute)   [65]
        - generate code to map attribute to byte sequence                     [66]
        - generate code to insert byte sequence into physical object referenced by poid-var-name[67]
```

For future work, line [66] will deserve special attention because it implicitly entails the knowledge on how to map complex values (e.g. LIST) to physical objects[1].

Now we show the contrary to physical object creation; the generation of code for physical object deletion given a reduction rules whose instantiation was triggered by DT operations. The only thing to do is to delete the physical object whose identifier is bound to the variable entailed in rr.PID of the actual reduction rule. In most of the cases, the variable will be a formal parameter of the current operation realization.

```
generate-delete-po(rr: fully-instantiated-reduction-rule)(                    [68]
```

1. As soon as there is a commitment on the interface of the underlying object server, the informally described algorithms can become more concrete.

```
        generate(delete-physical-object(rr.PID))                                  [69]
```

The third algorithm besides the already discussed generate-create-po and generate-delete-po is the algortihm generate-update-po which is applied in case of not fully-instantiated reduction rules at code generation time ([53], [58]). Not full-instantiation may happen if they DBI invokes a CT or DT operation where the given entity identifier is not created by a NEW operation but a formal parameter of the operation. In other words, the DBI attempts to work on an existing physical object. The attribute names showing up in those triple operations must occur in LIST or SET attribute assignements of complex value definitions of reduction rules (e.g. [10]). By the way, the last "must" defines the range of semi-instantiated reduction rules for which it is still possible to generate code.

```
generate-update-po(rr: partially-instantiated-reduction-rule)(                              [70]
    -- check whether PID is instantiated, proceed if so, else abort generation              [71]
    -- generate-code to locate the complex value within the physical object                 [72]
    -- for each instantiated complex value assignement of the reduction rule                [73]
    --      if rr is element of ctrr then                                                   [74]
    --              generate code which implements the assignement action (e.g. INSERT)[75]
    --      else //* rr is element of dtrr                                                  [76]
                    generate code which implements the inverse of the assignement action [77]
```

The next algorithm creates a function which delivers the offset where the value denoted by e.a can be found at runtime in case it is provided with a physical object identifier. The algorithm assumes that the attributes of the complex value specification are given by a list.

```
generate-extraction(rr: appropriate-reduction-rule; a: attribute)()                         [78]
            current:=car(rr.attributes)          /* get first attribute */                   [79]
            offset:=0                                                                        [80]
            WHILE current<> a (                                                              [81]
                    offset:=offset:=size(a)                                                  [82]
                    a:=cdr(att))                                                             [83]
```

Note that this algorithm assumes that the components are of fixed size. However, as already mentioned, this assumption is not always justified, e.g. if components are LIST- or SET-valued; even Strings are not of fixed size. Hence, the algorithm above has to be generalized in order to cope with these extended requirements. If one gives up the fixed-size approach, the layout of the physical objects which represent extensional information has to be carefully considered.

The only information which can be concluded from the reduction rule in the context of variable size attributes is the index of the attribute within the TUPLE. This index can be used to retrieve the actual offset from a table which is stored in a predefined place in the physical object. Whenever the value of an attribute changes its size, the corresponding entries in the table have to be updated. Since the number of attributes is generally of very limited size, the loss of storage space should not be any problem[1]. Furthermore, the loss of processing time is constantly small since the length of list is fixed and the indices are no matter of changes. Thus, an array would be an appropriate data structure to be basis of such a scheme since it is easy to access and to iterate over in order to update the offsets on behalf of particular changes.

```
generate-extraction2(rr: appropriate-reduction-rule; a: attribute)(        [84]
    generate-offset(attribute-index(rr, a)))                                [85]
```

The inner call returns the index of the attribute a within the reduction rule rr. Generatate-offset produces a function, which returns the position of the index(rr.attributes, a) ‘th attribute if the object identifier is passed as parameter.

The next algorithm to be considered in this context is the generate-replace algorithm which is parametrized by three variable names whose values are the physical object identifier, the offset at runtime and a reference to the new value that is intended to replace the old one. Furthermore the size and an index indicating the relative position of the component to replace are passed as parameters. The size of the new value and the index are necessary to recompute the offset table.

```
generate-replace(poid-var-name, offset-var-name, index, value-var-name, size)(        [86]
    - generate code to compute difference between the length of the new and old value  [87]
    - generate code to replace the old value (i.e. realize INSERT/DELETE for LISTS)     [88]
    - add the difference in the length to all entries in the offset table which are beyond index[89]
```

---

3. 2. 4 Reduction Rule Manipulation

Finally, the different reduction rule manipulation algorithms are specified as well. These algorithms assumes a global set of reduction rules to be stored in a particular structure which is elaborated on first.

```
match( set-expression, entity, attribute)                        [90]
instantiate(reduction-rule, entity, attribute, value)            [91]
attribute-index(reduction-rule, attribute)                       [92]
fully-instantiatedp(reduction-rule)                              [93]
```

1. Furthermore, note that the object server allows the insert and delete of arbitrary byte sequences at any place in the physical object. Thus, storage management is no matter of concern here.

4 An Example: The Specification of the NO^2 Data Model

To show the applicability of the meta data model, we will present the specification of the NO^2 data model [Dittrich 92] in this section. NO^2 has been chosen since, most of the people in our project know this particular data model well and KIDS becomes an official task within the ITHACA project. So in order to morally justify KIDS, CoOMS [Dumm 91] must be the example to show the overall feasibility of our approach.

In the following subsections, we will give the specifications of the NO^2 meta value sets, the meta object type, and the meta association types. Then these meta definitions are collected into a complete data model specification. Moreover, some representative results of applying the algorithms of section 3 applied to this specification are presented.

4. 1 Meta Value Sets

```
MCVS Set[ (VSid: INTENSION; Vid: EXTENSION) ]
    INTENSION IDENTIFICATION                          /* results in two new concrete types: */
        SURROGATE(SetVSid), String(Name)              /* SetVSid and Name */
    EXTENSION IDENTIFICATION                          /* results in one new contrete type: */
        SURROGATE(SetVid)                             /* SetVid */
    INTENSION INTERFACE                               /* now the DDL operations */
        Set(vsid: VSid; name: Name): SetVSid          /* primary constructor for set value sets */
        elementVS(vsid: SetVSid): VSid                /* the element value set of a set value set */
    EXTENSION INTERFACE                               /* now the DML operations */
        create(vsid: SetVSid): SetVid                 /* this creates a new set */
        delete(s: SetVid): Boolean                    /* this deletes the set */
        ∅(): SetVid                                   /* empty set constant */
        insert(s: SetVid; e: Vid): Vid                /* returns s with inserted e */
        remove(s: SetVid; e: Vid): Vid                /* returns s with removed e */
        union(s1: SetVid; s2: SetVid) : SetVid        /* returns s1 union s2 in s1 */
        intersection(s1: SetVid; s2: SetVid) : SetVid /* returns s1 int s2 in s1 */
        difference(s1: SetVid; s2: SetVid): SetVid    /* returns s1 -s2 in s1 */
        elementp(s: SetVid; e: Vid) : Boolean         /* true if e element of s */
        cardinality(s: SetVid): Integer               /* returns number of elems *./
        getVS(svid: SetVid): SetVSid                  /* returns the value set of a set value */
        select(svid: SetVid; f: Vid -> Boolean): Boolean
    INTENSION REALIZATION
        Set(vsid: VSid; name: Name) (
            svs:=NEW(SetVSid)                         /* creates new identifier */
            CT(SCHEMA, svs, s-name, name)             /* attach set name to it */
            CT(SCHEMA, svs, elem-vs, vsid)            /* attach element vs info to it */
            RETURN svs)
        elementVS(vsid: SetVSid) (
```

```
            RETURN svs.elem-vs)                    /* retrieval from “triple database” */
EXTENSION REALIZATION
    create(vsid: SetVSid) (
            sv:=NEW(SetVid)                        /* create new Set value identifier*/
            CT(DB, sv, vs, vsid)                   /* attach Set value set information */
            RETURN sv)                             /* return identifier */
    delete(sv: SetVid) (
            ι_s[λp. delete(p.elem)] ((σ[λt. ENT(t) =sv ∧ ATT(t)=elem](DB))
             /* deleted set element values*/
            DT(DB, sv, elem,_)                     /* unattach element triples */
            DT(DB, sv, vs,_)                       /* unattach type information */
            RETURN true)
    insert(sv: SetVid; ev: Vid) (
            IF ¬ Set.elementp(ev,vs) THEN CT(DB, sv, elem, ev) /* example for op reuse */
            RETURN s)
    elementp(sv: SetVid; ev: Vid) (
            RETURN sv.vs.elem-vs = ev.vs ∧         /* a rather complex triple query */
                    σ[λs. s.elem = ev](DB)≠ ∅)
    remove(sv: SetVid; ev: Vid) (
            IF elementp(ev,sv) THEN DT(DB, sv, elem, ev)  /* again a reuse of elementp */
            RETURN sv)
    union(s1: SetVid; s2: SetVid) (
            IF s1.vs.elem-vs = s2.vs.elem-vs THEN      /* check for element type compatib. */
                    ι_s[λp. Set.insert(s2, p.elem)]
                            ((σ[λt. ENT(t) = s1 ∧ ATT(s1) = elem](DB))
                    RETURN s2
            ELSE /* type compatibility error */ )
    difference(s1: SetVid; s2: SetVid) (
            IF s1.vs.elem-vs = s2.vs.elem-vs THEN
                    ι_s[λp. Set.remove(s2, p.elem)]
                            ((σ[λt. ENT(t) = s1 ∧ ATT(s1) = elem](DB))
                    RETURN s2
            ELSE /* type compatibility error */ )
     intersection(s1: SetVid; s2: SetVid) (
            s3:=s1
            s4:=s2
            s5:= union(s1, s2)
            Set.difference(s1, s4)
            Set.difference(s2, s3)
            Set.difference(s3, s5)
            Set.difference(s4, s5)
            RETURN s5)
 INTENSION REDUCTION
```

```
            SetValueSet: TUPLE(PID: SetVSid; evs: VSid; name: String)                    [71]
            SetValueSet:= [PID:=svs; evs:=svs.elem-vs; name:=svs.name]                   [72]
                  (σ[svs.  (TYPE(ENT(svs) = SetVSid) ∧                                   [73]
                                   ((ATT(svs) = name) ∨ (ATT(svs)=vs))](SCHEMA))         [74]
      EXTENSION REDUCTION                                                                [75]
            SetValue: TUPLE(PID: SetVid; vs: VSid; elements: SET(Vid))
                                                                                         [76]
            SetValue:= [PID:=sv; vs:=sv.vs; elements:= INSERT(elements, sv.elem)]        [77]
                  (σ[sv. (TYPE(ENT(sv) = SetVid) ∧ ((ATT(sv) = elem) ∨ (ATT(sv) = vs))](DB))   [78]
SVCM                                                                                     [79]
MCVS Tuple[ (VSid: INTENSION; Vid: EXTENSION) ]                                          [80]
      INTENSION IDENTIFICATION                                                           [81]
            SURROGATE(TupleVSid), String(Name)                                           [82]
      EXTENSION IDENTIFICATION                                                           [83]
            SURROGATE(TupleVid)                                                          [84]
      INTENSION INTERFACE                                                                [85]
            Tuple(attNames: List(String); attVSids: List(VSid); nm: Name): TupleVSid     [86]
            attNames(vsid: TupleVSid): List(String)                                      [87]
            attVSids(vsid: TupleVSid): List(VSid)                                        [88]
      EXTENSION INTERFACE                                                                [89]
            create(vsid: TupleVSid): TupleVid                                            [90]
            delete(vsid: TupleVSid): Boolean                                             [91]
            put(tv:TupleVid; attName: String; attVal: Vid) : TupleVid                    [92]
            get(tv: TupleVid; attName: String) : Vid                                     [93]
      INTENSION REALIZATION                                                              [94]
            Tuple(attNames: List(String); attVSids: List(VSid); nm: Name)(               [95]
                  tvs:=NEW(TupleVSid)
                  CT(SCHEMA, tvs, name, nm)
                  IF List.length(attName) = List.length(attVSids) THEN (                 [96]
                        n:=attNames                                                      [97]
                        v:= attVSids                                                     [98]
                        WHILE List.cdr(l) <> List.nil (
                              CQ(SCHEMA, tvs, att-name, List.car(n))
                              CQ(SCHEMA, tvs, att-vs, List.car(v))                       [99]
                              n:= List.cdr(n)                                            [100]
                              v:= List.cdr(v)                                            [101]
                        RETURN tvs)                                                      [102]
                  ELSE /* error */)                                                      [103]
            attNames(vsid: TupleVSid) (                                                  [104]
                  l:=List.make(String)                                                   [105]
                  ι[λ t. List.cons(t, l)]( σ[ENT(t) = vsid  ∧ ATT(t) = att-name](SCHEMA)  [106]
                  RETURN l)                                                              [107]
            attVSids(vsid: TupleVSid) (                                                  [108]
```

```
                l:=List.make(VSid)                                                        [109]
                ι[λ t. List.cons(t, l)]( σ[ENT(t) = vsid  ∧ ATT(t) = att-vs](SCHEMA)       [110]
                RETURN l)                                                                 [111]
        EXTENSION REALIZATION                                                             [112]
            create(vsid: TupleVSid) (
                tv:=NEW(TupleVid)
                CT(DB, tv, vs, vsid)                    /* attach type info. */
                                                        /* initialize value with nil */
                ι_s[λvs. CQ(tv, att-value, nil]
                    ((σ[λvs. ENT(vs) =vsid ∧ ATT(t) = att-name](DB))
                RETURN tv)                                                                [113]
            delete(tv: TupleVid) (                                                        [114]
                DT(DB, tv, vs, _)                       /* unattach type info */          [115]
                                                        /* delete attribute values */     [116]
                ι_s[λp. delete(p.att-value)] ((σ[λt. ENT(t) = tv ∧ ATT(t) = att-value](DB)) [117]
                RETURN true)                                                              [118]
            put(tv: TupleVid; attName:String; attVid: Vid) (
                IF ENT(tv) THEN (/* type checking should be done here */
                    i:=ind(tv.vs, name,attName,_)
                    UQ (DB, tv, att-value, attVid, i)
                    RETURN t)
                ELSE /*error */ )                                                         [119]
            get(tv: TupleVid; attName: String) (
                IF ENT(tv) THEN ( /*type checking should be done here */
                    i:=ind(tv.vs, name, attname,_)
                    RETURN VAL(tv,att-value,i,_))
                ELSE /* error */ )                                                        [120]
        INTENSION REDUCTION                                                               [121]
            TupleValueSet : TUPLE(PID: TupleVSid; name: String;                           [122]
                                    attNames: LIST(STRING); attVSs: LIST(VSid))           [123]
            TupleValueSet:= [PID:= ENT(tvs); name:=tvs.name;                              [124]
                            attNames:= INSERT(attNames, tvs.att-name, IND);               [125]
                            attVSs:=INSERT(attVSs, tvs.att-name,IND)]                     [126]
                (σ[tvs. (TYPE(ENT(tv) = TupleVSid) ∧ ((ATT(svs) = name) ∨ (ATT(svs) = att-name) [127]
                                                        ∨ (ATT(svs) = att-vs))](SCHEMA)
                            /* IND is the index of the quadruple triggering the reduction rule */ [128]
        EXTENSION REDUCTION                                                               [129]
            TupleValue : TUPLE(PID: TupleVid; vs: TupleVSid; attValues: LIST(Vid))        [130]
            TupleValue:= [PID:= ENT(tv); vs:=tv.vs; attValues:= INSERT(attValues, tv.attValue,IND)] [131]
                (σ[tv. (TYPE(ENT(tv) = TupleVid) ∧ ((ATT(sv) = att-value) ∨ (ATT(tv) = vs))]   [132]
SVCM                                                                                      [133]

MCVS Array[ (VSid: INTENSION; Vid: EXTENSION) ]                                           [134]
```

```
        INTENSION IDENTIFICATION Surrogate(ArrayVSid), String(Name)
        EXTENSION IDENTIFICATION Surrogate(ArrayVid)
        INTENSION INTERFACE
                Array(n: Integer; elemVSid: VSid; nm:Name): ArrayVSid
        EXTENSION INTERFACE
                create(vsid: ArrayVSid): ArrayVid
                put(a: ArrayVid; e: Vid): ArrayVid
                get(a: ArrayVid; n: Integer): Vid
        INTENSION REALIZATION
                Array(n: Integer; elemVSid: VSid; nm:Name)(
                        avs:=NEW(ArrayVSid)
                        CT(SCHEMA, avs, vs, elemVSid)
                        CT(SCHEMA, avs, length, n)
                        CT(SCHEMA, avs, name, nm)
                        RETURN avs)
        EXTENSION REALIZATION
                create(vsid: ArrayVSid)(
                        av:=NEW(ArrayVid)               /* new value id */
                        CT(DB, av, vs, vsid)            /* attach type info */
                        i:=vsid.length
                        WHILE i > 0 (                   /* initialize entries with N.A */
                                CQ(DB, av, entry, NA)   /* so the entries are not available yet */
                                i:=i-1))
                put(a: ArrayVid; e: Vid)()
                get(a: ArrayVid; n: Integer)()
        INTENSION REDUCTION
        EXTENSION REDUCTION
SVCM

MCVS List[ (VSid: INTENSION; Vid: EXTENSION)]
        INTENSION IDENTIFICATION Surrogate(ListVSid), String(Name)
        EXTENSION IDENTIFICATION Surrogate(ListVid)
        INTENSION INTERFACE
                List(elemVSid: VSid; nm:Name): ListVSid
                elemVSid(vs: ListVSid): VSid
        EXTENSION INTERFACE
                create(vsid: ListVSid): ListVid
                nil(): ListVid                          /* nil is a constant */
                length(l: ListVid): Integer             /* number of elements */
                cons(e: Vid; l: ListVid): ListVid       /* insert e at front of l */
                cons(l1, l2: ListVid): ListVid          /* appends l2 to l1 */
                car(l: ListVid): Vid                    /* head element of l */
                cdr(l: ListVid): ListVid                /* tail elements of l */
```

```
            INTENSION REALIZATION
            EXTENSION REALIZATION
            INTENSION REDUCTION
            EXTENSION REDUCTION
SVCM

```

The main issue with the specification of the value manipulation operation is whether they act in a destructive manner or whether the result is a newly created value. From an operation realization point, both alternatives are feasible. The former one results in the deletion of some few triples and the latter ends up with the creation of a new value based on an existing one.

Note that this distinction is not visible from the operation signature itself but has to be made explicit by means of comments.

4. 2 Meta Object Type

In NO^2 , only one kind of object type exists. Hence the singular in the subsection’s title.

```
MOT ObjectType[ (VSid: INTENSION; Vid: EXTENSION) ]
            IDENTIFICATION INTENSIONS
                  SURROGATE(OTid), String(Name)
            IDENTIFICATION EXTENSIONS
                  SURROGATE(Oid)
            INTENSION INTERFACE
                  ObjectType(vsid: VSid; nm: Name): OTid          /* create an object type */
                  ObjectType(superOTid:OTid; nm: Name;            /* create an object type by inherit. */
                                    extendedAttNames: List(String);
                                    extendedAttVSids: List(VSid);
                                    refinedAttNames: List(String);
                                    refinedAttVSids: List(VSid): OTid
                  getVS(otid: OTid): VSid                         /* get vsid of object type */
            EXTENSION INTERFACE
                  create(vid: Vid; otid: OTid): Oid               /* create object given a value */
                  create(otid: OTid): Oid                         /* create object with “empty” value */
                  delete(obj: Oid): Boolean                       /* delete object */
                  putValue(oid: Oid; vid: Vid): Oid               /* replace old value by new value */
                  getValue(oid: Oid): Vid                         /* get object value */
                  getOT(oid: Oid): OTid                           /* get object’s object type */
                  identityp(oid1, oid2:Oid): Boolean              /* test for object identity */
                  equalityp(oid1, oid2:Oid): Boolean              /* test for object equality */
            INTENSION REALIZATION
                  ObjectType(vsid: VSid;m: Name) (
                        ot:=NEW(OTid)                             /* get new identifier */
                        CT(SCHEMA, ot, vs, vsid)                  /* attach type info */
```

```
            CT(SCHEMA, ot, name, nm)                /* attach naming info */
            RETURN ot)
        getVS(otid: OTid) (
            h:= σ[λe. ENT(e) = otid](DB)
            IF CARD(h) = 1 THEN
                RETURN ENT((PICK(h)).vs
            ELSE /* object type identification error does not exist error */)
    EXTENSION REALIZATION
        getOT(oid: Oid): (
            h:= σ[λe. ENT(e) = otid](DB)
            IF CARD(h) = 1 THEN
                RETURN ENT((PICK(h)).ot
            ELSE /* raise object identification error does not exist error */)
        create(vid: Vid; otid: OTid) (
            IF otid.vs=vid.vs THEN (
                obj:=NEW(Oid)                   /* get new identifier */
                CT(DB, obj, ot, otid)           /* attach type info */
                CT(DB, obj, val, vid)           /* attach its value id*/
                RETURN obj)                     /* return oid */
            ELSE /* raise type mismatch error */ )
        delete(oid: Oid) (
            DT(DB, obj, ot,_))                  /* delete type inform. */
            delete(DB, obj.val)                 /* delete its value */
            RETURN true )
        putValue(oid: Oid; vid: Vid) (
            IF oid.ot.vs = vid.vs THEN (
                UT(DB, oid, val, vid)           /* attach new value id */
                RETURN oid)
            ELSE /* type error */ )
        getValue(oid: Oid) (
            RETURN oid.val
        identity(obj1, oid2: Oid) (
            RETURN equal(oid1, oid2))
        equality(oid1, oid2: Oid) (
            RETURN equal(obj1.val, obj2.val))
    INTENSION REDUCTION
        ObjectType : TUPLE(PID: OTid; name: String; vsid: VSid)
        ObjectType:= [PID:= ENT(ot); name:=ot.name; vsid:=ot.vs]
            (σ[ot.  (TYPE(ENT(ot) = OTid) ∧ ((ATT(ot) = name) ∨ (ATT(ot) = vs))](SCHEMA)
    EXTENSION REDUCTION
        Object : TUPLE(PID: Oid; ott: OTid; value: Vid)
        Object:= [PID:=ENT(o); ot:=o.ot; value:= o.val]
            (σ[o. (TYPE(ENT(o) = Oid) ∧ ((ATT(o) = val) ∨ (ATT(o) = ot))](DB)
```

```
        EXTENSION PREDEFINED
        PERSISTENCE PREDEFINED
TOM
```

4. 3 Meta Association Types

In the following, the NO^2 concepts *subobject* and *general reference* are modeled using meta association type specifications. Note that inheritance is not specified this way because, in NO^2 inheritance primarily has type derivation semantics, thus being more or less just a means to define types based on existing definitions. Semantics beyond that may can be implemented by modifying the triple implementations accordingly.

The following meta association type specification defines the *subobject* concept. Note that the specification is parameterized. This significantly increases the generality of this specification.

```
MAT Sub[ (OTid: INTENSION; Oid: EXTENSION) ]
        INTENSION IDENTIFICATION
                SURROGATE(SubATid) String(Name)         /* means to identify sub assoc types */
        INTENSION INTERFACE
            Sub(    fromotid, tootid: OTid              /* the assoc type constructor */
                    sharable, dependent: Boolean;
                    name: Name) : SubATid
            getSubATid(name: Name): SubATid             /* name to identifier mapper func. */
            getSubATids(fromOTid): List(SubATid)        /* get sub object types of fromOTid */
            getSupATids(toOTid): List(SubATid)          /* get super objecttype of toOTid */
        EXTENSION INTERFACE
            create(s: SubATid; fromoid: Oid; tooid:Oid): Boolean  /* create an subobject association */
            remove(s: SubATid; fromoid: Oid; tooid:Oid): Boolean /* delete an association */
            getSub(s: SubATid; fromoid: Oid): Oid       /* get subobject */
            getSups(s: SubATid; tooid: Oid): SetOid     /* get super objects */
        INTENSION REALIZATION
            Sub( fromotid, tootid: OTid ; sharable, dependent: Boolean; name: Name) (
                subatid:=NEW(SubATid)                   /* get new sub assoc type identifier */
                CQ( subatid,sub-participant, 1, fromotid)   /* attach participated object types */
                CQ( subatid,sub-participant, 2, tootid)
                CT( subatid, sub-sharable, sharable)    /* attach integrity constraint info */
                CT( subatid, sub-dependent, dependent)
                CT(subatid, sub-name, name)             /* attach name */
                RETURN subatid)
        EXTENSION REALIZATION
            create(s: SubATid; fromoid, tooid:Oid) (
                /* type checking etc */
                subaid=NEW(SubAid)                      /* get new sub assoc identifier */
                CT( subaid, sub-at, s)                  /* attach assoc type info */
```

```
            CQ subaid, sub-participant, 1, fromoid)       /* attach participated objects */         [284]
            CQ( subaid, sub-participant, 2, tooid)                                                  [285]
            RETURN some boolean)                                                                    [286]
    INTENSION REDUCTION                                                                             [287]
    EXTENSION REDUCTION                                                                             [288]
    PROPERTIES                                                                                      [289]
        FUNCTIONAL INJECTIVE                                                                        [290]
    PROPAGATION                                                                                     [291]
        BEFORE ObjectType.delete(oid) (                   /* the event specification */             [292]
            ot:=getOT(oid)                                /* type id of involved object into ot*/   [293]
            subATs:= getSubATids(getOT(oid))              /* get the subobject types of ot */       [294]
            WHILE List.cdr(subATs) <>List. nil() (        /* iterate on them */                     [295]
                s:=List.car(subATs)                       /* get first subobject type */            [296]
                soid:=getSub(s, oid)                      /* get the subobject of this type */      [297]
                remove(s, oid, soid)                      /* unestablish association */             [298]
                IF s.sub-dependent AND (NOT s.sub-sharable OR                                       [299]
                        Set.card(getSups(s,soid) = 0) THEN                                          [300]
                        (ObjectType.delete(soid)) /* propagate deletion */                          [301]
                subAts:=List.cdr(subAts)))                /* prepare for next iteration */          [302]
TAM                                                                                                 [303]
```

Of course, the reduction rules given in lines [286] to ?? represent a default reduction only, since one may wish to reduce in another way depending on the participating meta object types. Thus if the DBI wishes another storage strategy, he has to specify it after having instantiated the meta association specification. However, up to now, we did not include that into the language, since we feel that the instantiation of specification when finally composing the data model is processed by the specification tools in way that the specifications are expanded. So then the DBI has the chance to change this expanded specification, i.e. he may replace some reduction rules according to the application profile and depending on the meta object types passed as parameters to meta association type specifications.

The following specifications sketches the modeling of the *general reference*-notion.

```
MAT Ref[ (OTid: INTENSION; Oid: EXTENSION)]                                                         [304]
    INTENSION IDENTIFICATION                                                                        [305]
        SURROGATE(RefATid), String(Name)                  /* means to identify assoc types */       [306]
    PROPERTIES FUNCTIONAL INJECTIVE                                                                 [307]
    INTENSION INTERFACE                                                                             [308]
        Ref(fromotid: OTid; tootid: OTid) : RefATid       /* creates a reference assoc. type */     [309]
    EXTENSION INTERFACE                                                                             [310]
        create(r: RefATid; fromoid: Oid; tooid:Oid): Boolean  /* creates a reference association */ [311]
        delete(r: RefATid; fromoid: Oid; tooid: Oid): Boolean /* deletes a reference association */ [312]
        deRef(r: RefATid; fromoid: Oid): Oid              /* navigates in assoc direction */        [313]
        refBy(r: RefATid; tooid: Oid): Oid                /* navigates opposite to assoc. dir. */   [314]
    PROPAGATION ....                                                                                [315]
```

```
        INTENSION REALIZATION....                                                [316]
        EXTENSION REALIZATION....                                                [317]
TAM                                                                              [318]
```

At this place, a clever reader may ask himself how to model more complex associations like ER-relationships which may themselves carry a state. From our point of view, it is a bad idea to mix up different things. So from a data model specification point of view, we would just offer set-valued functions to map the extension of one object type to another. The following meta association type specifies such system-functionality[1].

```
MAT ER[ (OTid: INTENSION; Oid: EXTENSION)]                                       [319]
        INTENSION IDENTIFICATION                                                 [320]
            SURROGATE(ERATid), String(Name)      /* means to identify assoc types */  [321]
        PROPERTIES                                                               [322]
            RELATIONAL TRANSITIVE SYMETRIC                                       [323]
        INTENSION INTERFACE                                                      [324]
            ER(fromotid, tootid: OTid; min, max: INTEGER): ERATid                [325]
        EXTENSION INTERFACE                                                      [326]
            create(eratid: ERATid; fromoid: Oid; tooid: Oid): Boolean            [327]
            delete(eratid: ERATid; fromoid: Oid; tooid: Oid): Boolean            [328]
            map(eratidr: ERATid; fromoid: Oid): Set(Oid)                         [329]
            unmap(eratid: ERATid; tooid: Oid): Set(Oid)                          [330]
        PROPAGATION                                                              [331]
        INTENSION REALIZATION                                                    [332]
        EXTENSION REALIZATION                                                    [333]
TAM ER                                                                           [334]
```

4. 4 Summary

Asumming the above specification in the specification library, summarizing the NO^2 data model is almost for free.

```
DM NO2                                                                           [335]
    VALUE SETS                                                                   [336]
        Integer, Real, LongField, Char, String,                                  [337]
        Set[VALUE SETS], List[VALUE SETS], Tuple[VALUE SETS], Array[VALUE SETS]  [338]
    OBJECT TYPES                                                                 [339]
        ObjectType[VALUE SETS]                                                   [340]
    ASSOCIATION TYPES                                                            [341]
        Ref[OBJECT TYPES]                                                        [342]
        Sub[OBJECT TYPES]
MD                                                                               [343]
```

This is the top-level specification of our desired data model. This specification may be viewed

1. Note that this specification is for binary relationships only.

as a query to the library of existing specifications. The result of this query is then verified (and possibly modified, e.g. by re-realizing operations or by replacing reduction rules) by the DBI and then passed to the subsystem generation tool which is elaborated in more detail in the next section.

4. 5 Subsystem

In this subsection, we illustrate the subsystem which would have been generated according to the algorithm in section 3. We mainly focus on the presentation of the subsystem interface.

5 Conclusion

This section compares our approach of specifying data models with the other we are aware of. Moreover, the actual state of this work is precisely stated, and an outline of the next steps to be taken is given.

5. 1 Related Work

The general goal of our approach is to construct instances of a particular class of systems, i.e. we are interested in the efficient construction of systems which behave like data managers of DBMS using a specificational approach. Since the specification of such complex software systems is a general problem of software engineering, its solutions (e.g formal specification methods) to reasonably reduce this complexity have been considered whether they were applicable in our context as well. Besides these general approaches, more concrete ones exist in the area of eDBMS with similar objectives to our approach.

5. 1. 1 General Formal Specification Methods

In most of these approaches, the general idea is to describe the systems by specifying them as abstract machines which offer some structures and some functionality at their interface. The interfaces are abstractly specified using specification languages following the abstract data type approach. The interfaces usually consist of a set of operation signatures defining the arity, the parameter types, and the result type of the operation. The semantics of the interfaces are usually defined in terms of specifying axioms on the operations. This ends up in term rewriting if one tries to make such specifications executable.

OBJ [Goguen 84] is a typical representant of a general purpose specification language which nicely reflect the history of formal specification languages. As with many others, the beginning had been a purely abstract data type approach with term-rewriting semantics. People recognized soon that such a purist's approach makes it difficult to reuse specifications, or in other words, makes it difficult to avoid tedious re-specification of the same artifact again and again. The logical consequence had been attempts to increase reusability. One notable approach is the parameterization of specifications [Goguen 84] which, e.g., allows the specification of stacks without concretely specifying the element type of the stack. However, the set of potential actual specification parameters (which is a set of types) can be defined using constraints with respect to the operations provided by types passed as actual parameters. Another notable approach to derive new specification of existing ones is the introduction of inter-specification relationships like inheritance. A more generalized view of inter-specification relationships leads to algebras over specifications ([Bergstra 90], [Breu 91]). Another notable approach is to introduce relationships like *is-subtype-of* between the specified types. This lead to a unification of object-oriented and functional programming paradigms [Goguen 87]. OBJ3 [Goguen 91] and OOZE [Alencar 91] are well known representants of this approach.

General purpose specification languages like OBJ3 and OOZE are not feasible for our purposes due to several reasons:

- *term rewriting semantics*: Most of the considered specification languages implement specification by means of term rewriting which is not practicable in the context of a data manager. Although the languages had nice syntax, they would have to be re-implemented to serve as a basis to data model specification.
- *generality*: The generality of the general purpose languages prohibits to include knowledge which is data management specific into the language. Refer to [Baats 89] and [Lavazza 89] for experiences using general formal specification languages in the area of data model specification.
- *expressiveness*: The general purpose specification languages are designed to allow the specification of concrete types which represent (simple) real world artifacts. However, in our setting, the set of possible concrete types has to be modeled using meta types. Thus, the existing specification languages lack an additional level of abstraction, the meta level.

Thus, after several (painful) attempts using general purpose specification languages we decided to design our own language which is tailored to the requirements given in [Geppert 92a]. Refer to [Baats 89] and [Lavazza 89] for experiences using general formal specification languages in the area of data model specification.

5. 1. 2 Specific Data Manager Construction Methods

The previous subsection did explain, why we do not advocate a general purpose specification language for data management subsystem construction. This small subsection is indeed more DBMS related. We show, how our approach to data management subsystem construction differs to existing eDBMS approaches to data management subsystem construction. As already mentioned in section 1, the existing approaches roughly non-distinctly classify into toolkit systems, kernel systems, and generative systems.

In toolkit systems, the data management subsystem is realized using reusable program components from a given library. In EXODUS [Carey 86] for example, the DBI uses a special purpose programming language (called E which is some kind of persistent C++) to procedurally implement the subsystem. There is no way to *specify* the system on a higher level. Thus the EXODUS eDBMS is rather a candidate to implement our specification language than to serve itself as a specification language.

In kernel systems, the data management subsystem is realized on top of an abstract machine offering for examples nested relations like DASDBS [Schek 90]. Again, the subsystem has to be procedurally programmed in a procedural programming language. Furthermore, all the data model structures are to be mapped on nested relations. Although the operations on nested relations might be efficiently implemented, the mapping to nested relations may not be suitable in all situations. Thus the utilization of kernel systems implies too much implementation effort and implies inflexibility with respect to physical representation of objects.

In generative systems, the data management subsystem is realized by directly mapping a specification written using a particular specification language to an executable data management subsystem. Note that the EXODUS optimizer generator [Graefe 87] uses this approach. There exist two well-known systems which fully rely on the generative approach

- In the GENESIS approach ([Batory 88a], [Batory 88b]), the data manager subsystem software is vertically divided into layers. The concepts offered a each layer are modeled using the GENESIS DDL which offers *files* consisting of records and links connecting records of different files. Of course, the concrete schemas on the top layer represent conceptual information whereas the schemas on the lowest level represent the physical database schema. GENESIS introduces a means (transformation model) to abstract structures more concrete ones and to represent the information at the lowest level (unifying model). The sum of transformations allows the generation of a subsystem which directly maps concrete data structure to physical ones.
  Our approach significantly differs in that the generated subsystem offers a conceptual data model depending on the data model to be realized, whereas the GENESIS subsystems have a fixed conceptual data model. As a consequence, in the GENESIS approach it is impossible to specify application domain specific operations for the object types. Similar to the GENESIS approach, we allow for a precise specification of the mapping of conceptual structures to physical ones.
- In [Maryanski 86], [Hong 90] a data model compiler (DMC) is proposed, whose initial goal was to automatically produce object-oriented DBMS with generated parts consisting of a conceptual model, a database design tool, a graphical query language and a mapper between conceptual and physical objects. The data models to be specified are always based on the ER model and varying in their set of offered relationships. In [Maryanski 86], some data models are described which are potential candidates to be specified using DMC. However, the description of the DMC itself is not very clear. In [Hong 90], a meta model is proposed to represent object-oriented data models. Unfortunately, the ambitious goal announced in the previous paper had been reduced to the generation of conceptual schema design tools from data model specifications. So it is not anymore a data model compiler at all, but rather a (graphical) schema checker. Nevertheless, the meta model serves a tool to describe the semantics of object-oriented data models.
  Our approach in data model specification is somehow similar in that data models are specified using meta object types and meta association types. It significately differs in that the number of attribute types is not restricted to predefined ones but may also specified using the concept of meta value set. Our approach fundamentally differs in that the objective is not to generate conceptual schema design tools but rather data management subsystems. We claim that the priority of methodologies for the construction of such subsystems is higher than for according schema design tools. A schema design tool without a data manager will be probably useless, even if there are mappings to existing data models because they inherently imply inefficiency.

As a summary, our contribution promises a complete methodology which allows the rapid construction of sufficiently efficient data managment subsystems.

5. 2 Actual Status of Work and Future Work

The work done up to now is merely slide and paperwork, since only a prototype parser to check the examples for syntactical correctness has been implemented so far. Of course, the meta data model and its implementation will be subject of further refinements. However, it is important to find a commitment on the basic concepts of this meta model as soon as possible. This is important because the other aspects (like integrity and transaction management) partially depend on the specification language presented in this report. Note that the authors' future emphasis will be on the other aspects!

6 Bibliography

[Alencar 91] Alencar A., Goguen J. A.; OOZE: An Object-Oriented Z Environment; In: Proc. European Conf. on Object-Oriented Programming, Geneva 91; LNCS 512, Springer, Berlin, FRG, 91; pp. 180-199;

[Baats 89] W.E. Baats, L.M.G. Feijs, J.H.A. Gelissen: A Formal Specification of INGRES. In: [Wirsing 89];

[Batory 88a] Batory D. S.; Concepts for a Database System Compiler; TR-88-01 of Departement of Computer Science, University of Texas at Austin, TX; January 88;

[Batory 88b] Batory D. S., Barnett J. R., Garza J. F., Smith K. P., Tsukuda K., Twichell B. C., Wise T. E.; Genesis: An Extensible Database Management System; IEEE TOSE 14 (11), November 88; pp. 1711-1730; and in: Zdonik S. B., Maier D.; Readings in Object-Oriented Database Systems; Morgan Kaufman Publishers Inc., San Mateo CA, USA, 90; pp. 500-518;

[Bergstra 90] Bergstra J. A., Heering J., Klint P.; Module Algebra; JACM, 37 (2), April 90; pp. 355-372;

[Breu 91] Breu R.; Algebraic Specification Techniques in Object-Oriented Programming Environments; Springer LNCS 562, Berlin, FRG, 91;

[Carey 86] Carey M. J., DeWitt D. J., Frank D., Graefe G., Muralikrishn M., Richardson J. E., Shekita E. J.; The Architecture of the EXODUS Extensible DBMS; In: Proc. 1st Intl. IEEE Workshop on Object-Oriented Database Systems; IEEE, Pacific Grove, Asilomar, CA, USA, 86; pp. 52-65;

[Chen 76] Chen P. P.; The Entity-Relationship Model: Towards a Unified View of Data; ACM TODS, 1 (1), March 76; pp. 9-36;

[Dittrich 91] Dittrich K. R., Geppert A., Goebel V., Nittel S., Scherrer S.; KIDS: A Declarative Approach to DBMS Construction; In: Proc. Conf. on Database Research in Switzerland, Lausanne, October 91; pp. 176-186;

[Dittrich 92] Dittrich K. R., Geppert A., Goebel V., Scherrer S.; The Data Definition Language of NO2; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; January 92;

[Dumm 91] Dumm T., Gorchs T., Watzek M.; The CoOMS Persistent Object Server. Requirement Specification, Functionality, and Architecture; Project Report ITHACA.SNI.91.X.5#2, Siemens Nixdorf Informationssysteme, Berlin 91;

[Geppert 92a] Geppert A., Scherrer S.; On KIDS and Concepts: Requirements for a Data Model Specification Language; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; June 92;

[Geppert 92b] Geppert A., Scherrer S.; EAV: An Algebra-Based Framework for the Specification of Objects and Rules; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; May 92;

[Goguen 84] Goguen J. A.; Parameterized Programming; IEEE TOSE, 10 (5), September 84; pp. 528-543;

[Goguen 87] Goguen J. A., Meseguer J.; Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics In: Shriver B., Wegner P. (Eds.): Research Directions in Object-Oriented Programming. MIT Press Series in Computer Systems, Cambridge MA, USA, 87;

[Goguen 91] Goguen J. A., Winkler T., Meseguer J., Futatsugi K., Jouannaud J.-P.; Introducing OBJ; Preliminary TR, Programming Research Group, Oxford University, Oxford, England; August 91;

[Graefe 87] Graefe G., DeWitt D. J.; The EXODUS Query Optimizer; In: Proc. ACM SIGMOD 1987 Intl. Conf. on Management of Data, San Francisco CA, USA, May 87; ACM, New York NY, USA, 87; pp. 160-171;

[Hong 90] Hong S., Maryanski F.; Using a Meta Model to Represent Object Oriented Data Models; In: Proc. 6th Intl. IEEE Conf. on Data Engineering, Los Angeles CA, USA, February 90; IEEE Computer Society, Los Alamitos CA, USA, 90; pp. 11-19;

[Kiczales 91] Kiczales G., des Rivieres J., Bobrow D. G.; The Art of the Meta Object Protocol; Capter 5 &6 of “The Art of the Metaobject Protocol”; MIT Press; USA; 91;

[Lavazza 89] L. Lavazza, S. Crespi-Reghizzi: Algebraic ADT Specification of an extended relational algebra and their conversion into a working prototype. In [Wirsing 89];

[Maryanski 86] Maryanski F., Bedell J., Hoelscher S., Hong S., McDonald L., Peckham J., Stock D.; The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems; In: Proc. 1st Intl. IEEE Workshop on Object-Oriented Database Systems; IEEE, Pacific Grove, Asilomar, CA, USA, 86; pp. 73-83;

[Nittel 92] Nittel S.; The KIDS Object Server Interface; Technical Report, Forschungsbereich Datenbanktechnologie, Institut für Informatik, Universität Zürich; July 92;

[Schek 90] Schek H.-J., Paul H.-B., Scholl M. H., Weikum G.; The DASDBS Project: Objectives, Experiences, and Future Prospects; IEEE TOKADE, 2(1), 90

[Wirsing 89] Wirsing M., Bergstra J. A.: Algebraic Methods: Theory, Tools and Applications. LNCS 394, Springer, Berlin, FRG;