Naming Schemes and Name Space Management in the DBPL Persistent Storage System *

Joachim W. Schmidt Florian Matthes

Department of Computer Science
Hamburg University
Schlüterstraße 70
D-2000 Hamburg 13
schmidt@rz.informatik.uni-hamburg.dbp.de

Abstract

Database applications must be capable of dynamically creating large quantities of names based on a scheme that can be communicated to and shared by some user community. The paper concentrates on this particular aspect of the set- and predicate-based Data Base Programming Language DBPL. It re-interprets the key and view mechanisms of relational database systems as naming schemes by introducing the concept of a *selector* for user-defined associative naming of partitions of large data sets. The DBPL approach results in a novel interpretation of relations as scopes with variable extents of named and typed data objects. The consequences of associative naming schemes on typing and binding mechanisms are discussed and various extensions required by data-intensive applications are introduced. The paper also describes the system support that the DBPL implementation provides for efficient name space management.

1 Introduction

The choice of a naming scheme is particularly crucial for any system in which names have to be communicated between a variety of users and where the use of names may extend over large time intervals and spatially distributed locations. Database applications require additional naming support for bulk data identification and for the representation of dynamic relationships between individual data objects.

The main objective of this paper is to present the naming scheme of DBPL [SEM88], a persistent language that provides – in addition to conventional types inherited from its algorithmic kernel Modula-2 [Wir83] – an advanced data model. This model includes first-order predicates and large data sets ("relations") and aims for type completeness and orthogonality.

The DBPL naming scheme allows the dynamic creation of object names from values and guarantees the uniqueness of names within the scope of a relation. This approach leads to a novel interpretation of relations as scopes with variable extents of named and typed data objects, as well as to a re-interpretation of database views and queries as mechanisms for scope restriction and iteration.

The paper presents in detail the language support provided by DBPL for such naming schemes and the naming and binding mechanisms realized by the DBPL persistent object store. In addition, it discusses the impact of DBPL's specific predicate-oriented approach to bulk data handling on the overall architecture of the system, on its low-level storage structures, its concurrency control protocols and its ability to support distribution and schema evolution.

2 Language Technology for Large Data Sets

Computer languages introduce *names* as tokens to be bound to computational entities for the purpose of their identification. Traditionally, a *binding* consists of a name-value pair [Str67]. Bindings have been augmented

*This research was supported in part by the European Commission under ESPRIT BRA contract # 4092 (FIDE).

by a type component [BL84] and then by an indication whether the value is constant or mutable [AM88]. Such augmentations serve to restrict the use of entities to those computational contexts that preserve the entities' intended semantics – and to formally, and effectively, assure such semantics. Binding can be made at compile time (static binding) or during program execution (dynamic binding) [MAD87, Dea89].

A binding is always performed within a particular environment. The *scope* of a name determines those parts of a program where the name can be used (e.g. to define other bindings). In static scoping, the scope can be determined by a static analysis of program texts, whereas in dynamic scoping these parts may vary from one program execution to another.

2.1 Elementary Naming and Typing in DBPL

Starting from its definitional context, a programming language object, e.g. a set variable, can be introduced into a computational context (expression, statement) or a communication position (parameter) by using its name. It can be protected by analysing its type and mutability status. Computations are based on its value.

```
MODULE m1;
  TYPE Digits = SET OF [0..9];
  CONST allDigits = Digits{0..9};
  VAR d: Digits;
BEGIN
  d:= allDigits;
END m1.
```

The module m1 defines the following (value) bindings:

< name= *allDigits*, value= {*0,1,2,3,4,5,6,7,8,9*}, type= *Digits*, access= {*read*} >
< name= *d*, value= *?*, type= *Digits*, access= {*read, assign*} >

Programming languages provide naming and binding mechanisms not only for entities of base types (like int, bool, char) or atomic types (like strings, sets or files), but also for structured objects (e.g. of type record or array). Such (built-in) type constructors provide naming schemes for nested object components, e.g. named selection for record fields or indexed selection for array elements, and allow binding to homogeneous (arrays) or heterogeneous (records) component types.

```
MODULE m2;
  TYPE Mass = RECORD value: REAL; unit: (g, kg, lb) END;
  TYPE Color = (red, green, blue);
  TYPE PartRecType = RECORD name: String; mass: Mass; color: Color END;
  VAR firstPart, nextPart: PartRecType;
BEGIN
  firstPart.mass.unit:= kg;
END m2.
```

Module m2 defines nested name spaces with the following bindings:

< name= *firstPart*, value= *?*, type= *PartRecType*, access= {*read, assign*} >
< name= *firstPart.name*, value= *?*, type= *String*, access= {*read,assign*} >
< name= *firstPart.mass*, value= *?* , type= *Mass*, access= {*read,assign*} >
< name= *firstPart.color*, value= *?* ,type= *Color*, access= {*read,assign*} >
< name= *nextPart*, value= *?*, type= *PartRecType*, access= {*read,assign*} >

By allowing recursive application of these nested naming schemes, modern programming languages are able to identify and bind a variable and unlimited number of data objects with a fixed set of (statically declared) names.

```
TYPE PartList = POINTER TO RECORD r:PartRecType; next: PartList END;
VAR p: PartList;
```

< name= *p*, value= *?*, type= *PartList*, access={*read, assign*} >
< name= *p*↑*.r*, value= *?*, type= *PartRecType*, access= {*read, assign*} >

< name= $p\uparrow.next$, value= *?*, type= *PartList*, access= {*read*, *assign*} >
< name= $p\uparrow.next\uparrow.r$, value= *?*, type= *PartRecType*, access= {*read*, *assign*} >
< name= $p\uparrow.next\uparrow.next$, value= *?*, type= *PartList*, access= {*read*, *assign*} >
...

The rationale behind such a variety of naming and binding mechanisms is derived from application demands: manipulating descriptions of aggregated commercial records [Hoa68] requires different schemes than evaluating matrices in scientific computations. However, there are other issues involved in the design of a particular naming scheme that seem to be independent of the particular application area:

- Orthogonality: All language objects should be named in a uniform way [Lan66, DD79].
- Modularity: The consequences of name changes should be limited to well-defined scopes [L+77, BL84, Har88, Wir83]. Modularity is supported by nested scopes in block-structured languages or by compilation units with explicit import interfaces. A flat name space for "external" names as in C may serve as a counter-example.
- Abstraction: A key feature of a naming scheme is the possibility to introduce new bindings based on existing bindings by abstracting from selected aspects of a the original binding and thereby only revealing "partial" (e.g. type) information about declared objects [L+77, Car89, DCBM89]. A similar motivation underlies view mechanisms [CDMB90] in database systems.

Naming and binding mechanisms in standard programming languages have also been employed for bulk data handling in persistent and shared environments. DBPL [MS89] provides a built-in type constructor for the definition of large data sets (relations). The following declaration

```
TYPE PartRelType = RELATION name OF PartRecType;
VAR Parts: PartRelType;
```

introduces, for example, a binding between the relation name Parts, the relation type PartRelType and a mutable (and initially undefined) relation value.

< name= *Parts*, value= *value*, type= *PartRelType*, access= {*read*,*assign*} >

The relation value always has to be a set (in the mathematical sense) of objects of the element type (in this case PartRecType) with the additional constraint expressed by the invariant

$\forall p_1, p_2 \in value(p_1.name = p_2.name) \Rightarrow (p_1 = p_2)$

2.2 Expressions and Statements in DBPL

Names which are bound to values of type relation may appear in the "standard contexts" provided by an imperative programming language:

- on the left-hand-side of an assignment operator;
- in right-hand-side expressions;
- as value parameters (call by value);
- as variable parameters (call by reference).

DBPL also provides the means to refer to a value of a relation type without introducing a name. The following syntax denotes relations by enumerating their elements:

```
PartRelType{}                          (* the empty relation of type PartRelType*)
PartRelType{firstPart, nextPart}
```

< name= , value= {}, type= *PartRelType*, access= {*read*}>
< name= , value= {*firstPart*, *nextPart*}, type= *PartRelType*, access= {*read*}>

The type identifier in a relation constructor is necessary to type check relation expressions using the standard type equivalence rules of Modula-2. These are based on name equivalence and not on structural equivalence.

To evaluate relations in expressions, DBPL provides specialized set-oriented *query expressions* for relation types. There are three kinds of query expressions, namely boolean expressions, selective and constructive query expressions.

Quantified expressions yield a boolean result by evaluating a boolean predicate for all elements of a relation:

```
SOME p IN Parts (p.name = "Table")
ALL p IN Parts (p.mass.unit = kg)
```

Selective query expressions in a relation constructor select a subrelation of a single relation value:

```
PartRelType{EACH p IN Parts: p.mass.unit = kg}
```

creates a relation of type PartRelType that contains (copies of) all elements of the relation variable Parts that fulfil the selection predicate p.mass.unit = kg.

Constructive query expressions construct relations based on the values of other relations:

```
TripleRelType{{pmin.color, pmin.price, pmax.price} OF
  EACH pmin IN Parts, EACH pmax IN parts:
    (pmin.color = pmax.color) AND
    ALL p IN Parts ((p.color <> pmin.color) OR
       (p.price >= pmin.price) AND (p.price <=pmax.price))}
```

The constructive query expression above defines how to derive a relation consisting of triples from the relation Parts. Each triple consists of the color of a part and the minimal and maximal costs for parts of that color. The type of the result relation is given by

```
TYPE TripleRelType = RELATION OF RECORD c: Color; min,max: REAL END;
```

In general, the evaluation of a constructive query expression yields a relation that contains the values of the target expression (preceding the keyword OF), evaluated for all combinations of the element variables (pmin, pmax) that fulfil the selection expression (pmin.color = pmax.color) AND ALL ...

To simplify the manipulation of large data sets, DBPL offers specialized relational update operators that are equivalent to more complicated relation assignments:

```
parts:+ newParts;                (* relation insertion *)
parts:= PartRelType{ EACH p IN parts: TRUE,
                     EACH np IN newParts: NOT SOME p IN parts (np.name = p.name)}

parts:- newParts;                (* relation deletion *)
parts:= PartRelType{ EACH p IN parts: NOT SOME np IN newParts (np.name = p.name)}

parts:& newParts;                (* relation update *)
parts:= PartRelType{ EACH p IN parts: NOT SOME np IN newParts (np.name = p.name),
                     EACH np IN newParts: SOME p IN parts (np.name = p.name)}
```

2.3 Scopes and Lifetime in DBPL

Depending on the specific demands of an application area, bindings may include additional attributes that regulate, for example, object lifetime, sharability, recovery, clustering, mobility and object access rights (see also Section 3.3).

In general, each of these additional attributes can be determined at various points in time and at different levels of granularity. As is the case with type judgements [Mat87], it is often preferable to trade in the flexibility of fully dynamic modifications of these object properties for the simplicity and efficiency of an early, mostly static decision at object declaration or creation time. Similarly, many binding decisions are not taken on a per-object basis but for composite objects or for entire scopes like modules or directories.

In addition to non-shared, transient variables as found in Modula-2, DBPL allows the declaration of shared and persistent database variables. This is achieved by extending the concept of a module to *database modules* (i.e. name spaces subject to separate compilation). All variables declared within a database module are persistent, i.e. in contrast to program variables in other modules their lifetime exceeds a single program execution. To be precise, the lifetime of a persistent variable is longer than that of any program importing it.

Persistent variables are shared objects and can thus be accessed by several programs simultaneously. An access to a persistent variable must be part of the execution of a transaction [MRS84, RS81]. Database modules can therefore be viewed as self-contained, statically declared, persistent, serializable and recoverable scopes.

In the operations explained above it is sufficient to treat relations as entire indivisable objects that can be named and typed without any visible substructure. Neglecting some syntactic pecularities (listfix notation for relation constructors or infix notation for some operators) the built-in relation types of DBPL could therefore be easily implemented by means of generic, parameterized abstract data types in a sufficiently rich type system. In Quest [Car89], the signature of a generic relation type with similar operations as in DBPL could be defined as follows. (Note that the create operation has an equality predicate as an additional parameter that is needed to maintain the key uniqueness in subsequent operations on the relation.)

```
interface Relation
export
  RelType::ALL(ElemType::TYPE) TYPE          (* a type operator *)

  create(A::TYPE elem:Array(A) equality(:A :A):Bool) :RelType(A)

  :==, :+, :-, :& (A::TYPE var lhs :RelType(A) rhs :RelType(A)) :Ok

  some,all(A ::TYPE range :RelType(A) predicate(:A) :Bool) :Bool

  each(A::TYPE range :RelType(A) predicate(:A) :Bool) :RelType(A)

  each2(A,B,C::TYPE range1 :RelType(A) range2 :RelType(B)
            predicate(:A :B) :Bool projection(:A :B) :C) :RelType(C)
end;
```

As it turns out, this view of a bulk data structure as an "abstract" named entity is too coarse for data-intensive applications requiring naming schemes which support the identification of specific subsets or individual elements of a collection.

3 Extended DBPL Naming Schemes

The previous section outlined predefined standard naming schemes in DBPL that enable programmers to declare and manipulate relation variables. The following subsections will study three extensions of these naming schemes that are mainly motivated by the particular needs of data-intensive applications in a persistent multi-user environment:

- Refinement of the granularity of values that can be named and bound;
- Language support for user-defined parameterized naming schemes;
- Extension of the set of attributes that can be bound to a name.

3.1 Key-Based Naming Schemes

The uniqueness constraint on relation key values expressed by the constraint

$\forall p_1, p_2 \in value(p_1.name = p_2.name) \Rightarrow (p_1 = p_2)$

allows the extension of the traditional notion of relations as value spaces (bulk data structures). It also permits the viewing relations as name spaces consisting of sets of bindings between key values and their associated relation elements.

For example, the two-element relation constructed with

```
Parts:= PartRelType{ {"Table", {10.0, kg}, red}, {"Chair", {2.0, kg},blue} }
```

defines the following bindings

< name= *Parts*, value= {*firstPart*, *nextPart*}, type= *PartRelType*, access= {*read*,*assign*} >
< name= *Parts["Table"]*, value= {*"Table"*, {*10.0, kg*}, *red*}, type= *PartRecType*, access= {*read*, *assign*} >
< name= *Parts["Chair"]*, value= {*"Chair"*, {*2.0, kg*}, *blue*}, type= *PartRecType*, access= {*read*, *assign*} >

The syntax of DBPL therefore allows (analogous to the selection of array elements) a denotation of relation elements by a relation name followed by a key value in square brackets. Parts["Table"] is a name for the relation element with the key value "Table" in the relation Parts. Note that Parts["Table"] does not simply denote a copy of a relation element, but that it is a name for an updateable subcomponent of the relation variable Parts (reference semantics). These names can therefore be used on the left-hand-side of an assignment operator, in right-hand-side expressions, as well as value and variable parameters. These are examples of selected relation element variables:

```
InOut.ReadReal(Parts["Chair"].mass.value);
Parts["Table"].mass.value := Parts["Chair"].mass.value * 4;
```

The built-in key selector that comes with every relation type may be viewed as a generic *naming scheme* that creates an object name for a given object value:

```
"Table" → Parts["Table"]
"Chair" → Parts["Chair"]
```

This is a generic naming scheme since DBPL (as a consequence of the principle of type-completeness) allows the definition of relation types with arbitrary element types, lists of key components and key components of simple or constructed DBPL types[1].

```
VAR Suppliers: RELATION name, address OF
      RECORD name: String; address: RECORD city, country: String END;
         employees: RELATION OF String;
      END;
VAR Table: RELATION key OF RECORD key, value: INTEGER END;
VAR Grid: RELATION [1],[2] OF ARRAY [1..3] OF Real;
TYPE SparseVector = RELATION col OF RECORD col:INTEGER; val: REAL END;
VAR SparseMatrix: RELATION row OF RECORD row: CARDINAL; val: SparseVector END;

Companies["Bayer","Leverkusen","Germany"]
Table[i]
Grid[1.75, 1.9]
SparseMatrix [1342] [99].val
```

These examples illustrate that key selectors permit the definition of *problem-oriented* naming schemes which support the identification and manipulation even of individual attributes of deeply nested structures. The set-oriented operations presented in Section 2.2 can now be re-interpreted as operations on name spaces that add, remove, update und query sets of bindings and, in addition, guarantee the uniqueness of names (based on keys) within a given name space (relation value).

A selection based on a non-existent key value (Parts["Cupboard"]) or attempts to change the key value of a selected relation element variable Parts["Table"].name:= "Chair" are illegal and will be detected at runtime leading to a transaction abort. A formalization and generalization of the underlying constraints on selected relation elements will be given in Section 3.2.

[1] The current implementation of DBPL does not support relations as key components. This restriction is based on purely engineering decisions, since the semantics of "deep equality" are well understood.

The main conceptual and pragmatic advantages of a value-based naming scheme can be summarized as follows:

- A rich set of value constructors (records, arrays, sets, ...) implies an equally rich set of naming schemes.
- The unrestricted availabilty of dynamic creation and deletion mechanisms for names allows programmers to implement their own centralized or decentralized, traced or untraced, static or dynamic naming policies.

  However, this freedom can also become a burden, as soon as it becomes necessary to trace the propagation of names to determine whether a name can be safely deleted or not. This deletion can take place when no other objects contain references to a name. As it turns out, DBPL greatly simplifies this task by supporting arbitrary nested quantified predicates. For example, the following update operation deletes all suppliers that no longer supply any part:

  ```
  Suppliers :- SupplierRelType{EACH s IN Suppliers: NOT SOME p IN Parts (p.supplier = s.name)};
  ```

  Using recursive query expressions as supported by recursive constructors in DBPL [MRS89], it is possible to extend this approach to transitive reachability rules.
- "Printable" names are a must for the communication of identifiers across system boundaries (e.g. at the user-interface or for data exchange between autonomous persistent stores).
- A crucial point observed in long-lived information systems is the fact that values can be "detected" to be names (e.g. the ZIP code of a city used in an early version of a shipment database can be used to obtain additional attributes of a city in a later version of the system) or uniqueness assumptions about names cease to be valid (e.g. two persistent name spaces need to be merged and duplicate names have to be identified).

In a language that does not provide an associative naming scheme, programmers usually resort to one of the following standard solutions to identify elements in a bulk data structure:

Explicit Navigation: In a first step, an "invisible" cursor is positioned within the bulk data structure and subsequent update and read operations operate implicitly on the selected element. As soon as there is a need to manipulate several cursors, this approach becomes impractical because of the overhead to open, close and identify the cursors themselves [Dat89].

Separate Get and Update Operations: Every access to an element of the bulk data structure has to pass a key value as an additional parameter. Update operations tend to be inefficient because of the duplicate key lookup in the get and update operation. Particular problems arise in the selection of deeply nested components since unnecessary large intermediate structures have to be retrieved and locked. For example, a simple operation like INC(SparseMatrix [1342] [99].val) has to be implemented as

```
VAR row1342: SparseVector;
VAR element: REAL;
row1342:= GetByIntKeyvalue(SparseVector, 1342);
element:= GetByIntKeyvalue(row1342.val, 99);
INC(element.val)
PutByIntKeyvalue(row1342.val,99, element);
PutByIntKeyvalue(SparseVector, 1342, row1342);
```

Indirect Access via References: The bulk data structure simply consists of an index that maps key values to references which point to the objects of the element type. All read and update operations are performed using references. The main disadvantage of this approach is the unrestricted scope and lifetime of the references. The uncontrolled proliferation of identifiers prohibits efficient storage management, access control, concurrency control, and recovery strategies to be utilized for element-oriented access.

3.2 Association-Based Naming Schemes

A serious shortcoming of the key-based naming scheme is its restriction to single-element access. What is still missing is a value-based naming mechanism with a granularity that mediates between full relation variables and individual relation elements. A first step towards a generalization of the key selection mechanism is the observation that in an expression context the following equality holds:

PartsRelType{Parts["Table"]} = PartsRelType{EACH p IN Parts: p.name = "Table"}

A key-based selector can therefore be viewed as an abbreviation for a specific parameterized selective query expression (see Section 2.2), EACH p IN Parts: p.name = keyvalue. The following *selector* declaration makes the underlying naming mechanism explicit:

```
SELECTOR ByKey ON (P: PartRelType) WITH (keyvalue: String): PartRecType;
BEGIN EACH p IN P: p.name = keyvalue END ByKey;
```

Given this declaration, the selector ByKey declares a parameterized naming scheme that maps from key values to names and Parts[ByKey("Table")] is the name for the (unique) element of type PartRecType in the relation Parts that fulfils the selection predicate p.name = keyvalue. The two names, Parts["Table"] and Parts[ByKey("Table")], always refer to the same relation element since the former is considered a shorthand for the latter.

Since the syntax of selective access expressions already allows a much richer set of selection predicates to be used in place of x.keyattribute = keyvalue, the generalization of key-based to general predicate-based naming schemes is straightforward.

```
SELECTOR ByColor ON (P: PartRelType) WITH (colorvalue: Color): PartRelType;
BEGIN EACH p IN P: p.color = colorvalue END ByColor;
```

The selector ByColor allows the selection of a *set* of parts with a given value for the color attribute: Parts[ByColor(red)] is therefore a name for that subrelation of the relation variable Parts, that contains only those elements that fulfil the selection predicate p.color = red. Note that the result type of the selector ByColor is PartRelType and no longer PartRecType. This reflects the fact that the selection yields a set of parts, i.e. there is no implicit de-setting operation as in the case of the selector ByKey.

Again, such *selected relation variables* can be used like plain relation names in expressions and assignments, and as parameters for all standard and user-defined procedures and functions. The semantics of the use of selected relation variables in such contexts can be deduced from the semantics of selected relation assignment. Since the formal definition of selective set assignment [Sch88] goes beyond the scope of this paper, only an intuitive presentation of DBPL selector semantics is provided.

In its most general form a selector declaration can be based on arbitrary selective access expressions:

```
SELECTOR sp ON (Rel: RelType) WITH ( parameter list): RelType;
BEGIN EACH r IN Rel: p(r, parameter list) END sp;
```

The goal of assigning an expression, rex, selectively to a relation, i.e., R[sp(...)] := rex, is to substitute the selected part of R by the r.h.s.-expression, rex, while keeping the non-selected part of the l.h.s.-variable invariant with respect to its pre-value, 'R. In terms of non-selective assignments we aim for a post-value, R', such that

R'[sp(...)] = rex and
R'[*not*sp(...)] = 'R[*not*sp(...)]

This goal can be reached by an implementation such as

R := {EACH r IN rex: TRUE, EACH r IN R: NOT p(r,...)}

However, as already demonstrated in Section 3.1 through examples of key-based selectors, this implementation does not consider possible constraint violations. In the example

Parts["Table"] := PartRecType{ PartName, {5.0, kg}, blue}

we do not intend to replace tables by chairs and therefore have to protect the assignment by an appropriate conditional:

```
IF PartName = "Table" THEN
  Parts:= {{PartName, {5.0, kg}, blue}, EACH p IN Parts: p.name<>"Table"}
ELSE exception END
```

Without going into a deeper case analysis, this semantics can be shown to generalize to conditionals that are universally quantified over arbitrary r.h.s. set expressions rex and to arbitrary first-order selection predicates p.

The signature of the above declared selectors defines a name for the selector, an ON-parameter that defines the range relation for the selector and a WITH-parameter that is used in the selection predicate. The ON-parameter allows the application of selectors to different (selected) range relations of a given type, for example

```
VAR OldParts: PartRelType;

OldParts[ByColor(red)]
OldParts[ByColor(red)] [ByKey("Table")]
```

It is possible to omit this parameterization:

```
SELECTOR RedParts: PartRelType;
BEGIN EACH p IN Parts: p.color = red END RedParts;

OldParts:= PartRelType{EACH p IN [RedParts]: p.mass.unit = lb};
```

The last example emphasizes the similarity between the use of selectors in DBPL and the use of *views* in database systems. Both allow the derivation of new named collections from other collections (or views) by means of value-based restrictions that abstract from any detail of the selection process.

The previous examples only make use of propositional predicates. However, quantified predicates are also admissable in selector declarations and provide mechanisms for bulk data identification based on *dynamic relationships* that involve names of different name spaces:

```
SELECTOR SuppliedFrom ON (P: PartRelType) WITH (city: String): PartRelType;
BEGIN
  EACH p IN Parts: SOME s IN Suppliers ((s.name = p.supplier) AND (s.city = city))
END SuppliedFrom;

Parts[SuppliedFrom("Hamburg")]
```

An important application for selectors is the enforcement of referential integrity between multiple name spaces. The basic idea is to define selectors on relation variables that only select "consistent" elements, i.e. those that fulfil the referential integrity constraint. The semantics of selective relation assignment then guarantee that updates using the selector will be rejected if they violate the integrity constraint.

```
SELECTOR ConsistentParts: PartRelType;
BEGIN EACH p IN Parts: SOME s IN Suppliers (s.name = p.supplier) END ConsistentParts;
```

According to the semantics of selected relation updates sketched above, the selected assignment [ConsistentParts] := OldParts is equivalent to the following conditional unselected assignment:

```
IF ALL p IN OldParts SOME s IN Suppliers (s.name = p.supplier) THEN
  Parts:= OldParts;
ELSE exception END
```

To summarize, the use of selectors for the definition of parameterized and value-based naming schemes, for scope restriction and for constraint maintenance has been presented. Since selectors are named and typed language objects, they can be stored as components of arbitrary data structures or passed as parameters to procedures or other selectors. [RM87] illustrates how the use of (non-parameterized) query expressions as first-class language citizens allows the user to simulate complex objects and to avoid explicit joins to materialize object relationships. DBPL separates parameter substitution and application of selectors (i.e. to "curry"selectors) . The latter is especially useful in conjunction with the module concept of DBPL and allows

the refinement from parameterized and unbound (permissive) to non-parameterized and bound (restrictive) selectors to take place in different scopes.

Finally, it should be noted that DBPL generalizes the concept of selectors (which are based on selective access expressions) to constructors that abstract over arbitrary constructive or selective access expressions (see Section 2.2). Recursive constructor definitions have fixed point semantics and thereby integrate the essence of logic-based query languages in a strongly typed and imperative framework [MRS89]. By storing sets of constructors in persistent relation variables, it is possible to arrive at a uniform (dynamic) management of data and rules [SL85, SGLJ89].

3.3 Extended Binding Attributes

In persistent, large-scale and shared environments it becomes necessary to extend the classical notion of a binding (see Section 2) by attributes that describe additional important object properties (like lifetime, accessability or site in distributed environments). Since there is a wide range of binding alternatives (like binding time and binding granularity) the following paragraphs will only highlight the specific decisions made in the DBPL language and system, and compare them with selected approaches of other PPLs and DBPLs.

3.3.1 Object value constraints

A first extension of binding attributes was already introduced in the previous sections: Relation elements denoted by a key selector application and selected relation variables are associated with object value constraints expressed by (first-order) predicates that have to be preserved on updates:

```
< name= Parts["Table"], value= {"Table", ... }, type= PartRecType, access= {read, assign},
      assertion= (Parts["Table"].name = "Table") >
< name= Parts[SuppliedFrom("Hamburg")], value= ..., type= PartRelType, access= {read, assign},
      assertion= (ALL p IN Parts[SuppliedFrom("Hamburg")]
            SOME s IN Suppliers ((s.name = p.supplier) AND (s.city = city))) >
```

A similar mechanism for the declaration of object value constraints can be found in Galileo [AGO89] and enables programmers to attach assertions (boolean-valued functions) to instances of classes. However, these assertions are only checked at instance creation time.

3.3.2 Object lifetime

Modern languages infer the lifetime of an object from the scope of its name(s). Typical examples are local objects in block structured languages that are created at block entry and cease to exist at block exit time, and dynamically allocated objects on a heap that are managed according to a transitive reachability rule [Coh81]. Persistent programming languages apply these rules uniformly to transient and persistent data objects [But87, ACC81, Bro89, Car86].

As illustrated in Section 3.1 and 3.2, DBPL provides in addition to these automatically managed scopes other naming schemes that decouple object lifetime from object naming and allow, in addition, explicit object creation and deletion. As a consequence, these naming schemes do not enforce that every name in a scope be bound to an associated object value.

3.3.3 Object sharability

In an environment where multiple threads of control access the same scope (e.g. in a multi-user environment), it becomes possible to attach different values to the same name, depending on the dynamic context in which the name is used. This is the approach taken by conventional programming languages, where each invocation of a program is equipped with its own, private global variables.

The notion of multiple programs accessing a common shared database can be captured in a language context by interpreting databases as scopes for global names that are bound to non-replicated values and

that are furthermore associated with specialized concurrency control mechanisms that guarantee serializable schedules for concurrent read and write accesses.

DBPL uses standard transaction-oriented concurrency control mechanisms on all persistent variables in all database modules. However, if different modules implement data structures at different levels of abstraction, it becomes useful to have a finer control of the access control mechanisms (unrestricted access, mutual exclusion, monitor-like protocols, ...).

3.3.4 Object consistency and recovery

In persistent systems it is particularly useful to be able to define the notion of a "consistent" view of a name space. In DBPL (like in conventional database systems) this is accomplished by first defining an initial consistent state. All states that can be derived by applying a transaction to a consistent state are considered to be consistent themselves [Gra81, RS81, SWBM89].

A more implementation-oriented consistency and recovery mechanism can be found in languages based on the concept of a persistent heap. They support atomic and durable state transitions by means of a built-in commit or stabilize [Bro89] operation that affects the whole uniform persistent store and all active processes.

Since DBPL provides a (mostly) static partitioning of the store into a multiplicity of persistent and volatile name spaces (modules), the effects of recovery mechanisms (e.g., automatic UNDO of the effects of partially executed transactions in case of system failures or automatic REDO in case of transaction aborts caused by deadlocks) can be limited to specific name spaces. In the current implementation of DBPL, only persistent variables are subject to automatic recovery mechanisms. More experience is required to evaluate the advantages and disadvantages of this particular binding alternative.

3.3.5 Object clustering, placement and mobility

The performance of storage managers of large, disk-based address spaces can be dramatically improved by exploiting knowledge about the access patterns of programs running against the store. In addition to dynamic clustering algorithms, a primary source of static information about the locality or "relatedness" of data objects is the scope in which their names are visible and accessible. The particular mapping from scopes in DBPL to address spaces of the storage manager is described in more detail in Section 4.1.

In the distributed version of DBPL [JLRS88, JGL+88], database modules are (dynamically) bound to (local- or wide-area) network nodes that serve as repositories for the persistent data. The import of procedure, transaction or variable names from remote hosts allows the unification of a wide range of database access mechanisms like remote procedure call, database procedures (sagas [GMS87]), or database access with full distribution transparency, by means of a single language concept, the export of names from autonomous scopes.

3.3.6 Object access rights

Access rights may depend on a subject, an object and an operation performed by the subject on an object. Operating systems usually distinguish between read, write and execute operations on persistent or volatile data objects. [AM88] argues for separate access rights for read und write access to named and typed locations in a persistent store. For relation variables it is possible to distinguish read, assign, insert, delete and update operations.

In the construction of large scale database applications written in DBPL, selectors can be utilized to define at the module interface value-based access restrictions on database relations [MRS89]. By hiding the database relations in a local scope, clients of such a module are forced to perform updates and read operations through the views provided by the exported selectors. As it turns out, it is often useful to attach additional *access restrictions* to selectors, e.g. to allow only insert and read operations on a given data set. For example, take the integrity constraint ALL p IN Parts SOME s IN Suppliers (s.name = p.supplier) that can be violated only by insertions into the set of parts or deletions in the set of suppliers. DBPL therefore allows the user to attach statically access restrictions to selector declarations. These access restrictions are checked at compile-time (by a straight-forward extension of the type checker).

```
SELECTOR SecureSuppliers FOR (==, :&, :+): SupplierRelType;
BEGIN EACH s IN Suppliers: TRUE END SecureSuppliers;

[SecureSuppliers]:+ SupplierRelType{...};    (* OK *)
[SecureSuppliers]:- SupplierRelType{...};    (* compile-time error *)
```

4 Name Space Management in DBPL

This section describes how the design and implementation of the DBPL persistent store was influenced by the particular choice of naming and binding mechanisms present in DBPL.

In the DBPL project there is a strong commitment to implementability. A multi-user DBPL system under VAX/VMS is used at the Universities of Frankfurt and Hamburg since 1985 for lab courses on database programming. Based on this continuing implementation effort, there exist several prototype extensions for concurrency (optimistic, pessimistic and mixed strategies) [BJS86], integrity [Böt90], storage structures for complex objects, recursive queries [JLS85, SL85] and distribution [JLRS88, JGL+88] . The construction of a distributed DBPL system is based on ISO/OSI communication standards and involves a re-implementation of the DBPL compiler to generate native code for the IBM-PC/AT.

During the last year a substantial effort was made to integrate the experience gained with these prototype extensions into a new, portable implementation of the DBPL runtime system [SBK+88a] under VAX/VMS and Unix. Furthermore, a current project aims to interface the DBPL compiler to Suns optimizing compiler backend thereby enhancing the performance and interoperability of the DBPL system.

The overall architecture of the DBPL system [SBK+88b] is shown in Figure 1.

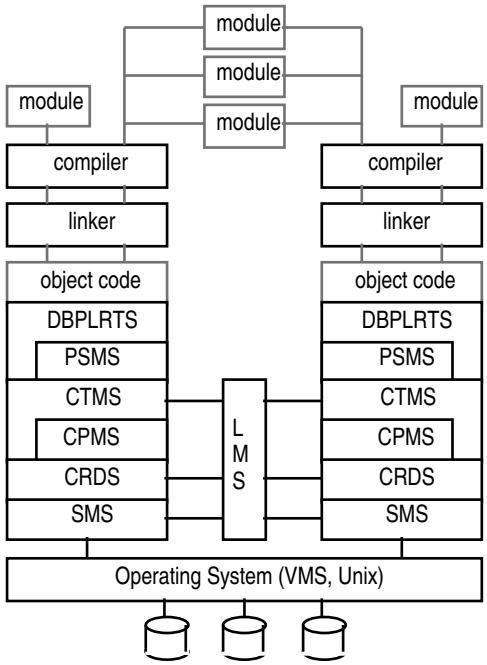

Figure 1: Architecture of the DBPL system

The DBPL compiler translates DBPL modules into object code of the target computer. A program may be comprised of several modules. Many constraints like type compatibilities and access restrictions are checked during compilation. In the object code, most of the DBPL extensions are realized by calls to the upper layer of the runtime system (DBPLRTS).

The layer DBPLRTS represents the interface to the runtime system, thereby abstracting from the internal representation of the objects defined within it. It provides support for iterations and is responsible

for the access to persistent variables that are not of type RELATION. Each executing DBPL object program is (conceptually) equipped with its own copy of the runtime system. Multiple executing DBPL programs (on a single machine or on different nodes in a LAN) communicate with a centralized concurrency control process (LMS).

The layer PSMS (predicate set management system) treats predicates (i.e. query expressions) as first-class objects. For selectors and constructors, it performs parameter substitution, expansion of nested predicates, and binding to program variables. Recursive constructors are computed by repeated evaluation of relation-valued expressions according to a fixpoint algorithm.

The layer CTMS (complex transaction management system) is responsible for concurrency control at the abstraction level of DBPL expressions and statements. According to the concept of multi-level synchronization [BSW88], it abstracts from lower-level conflicts arising from implementation details of storage structures or access path support provided by the layer CRDS and SMS.

The layer CPMS (complex predicate management system) implements data structures for the internal representation of queries over NF^2-relations. Internally, queries are represented by augmented syntax trees. They are transformed to simplify the subsequent evaluation. Set-oriented read and write operations are resolved into operations on complex relation elements.

The layer CRDS (complex relational data system) provides a navigational and element-oriented access, based on logical access paths. It contains means for synchronization and recovery, which are realized by primitives provided by the SMS.

On the basis of a block-oriented access, the storage management system SMS offers, in addition to the record and page management, page-overlapping long fields as data objects. The direct support of long fields on the storage level enhances the efficient implementation of complex objects and their access paths.

4.1 Dynamic Bulk Binding and Access Path Management

A fundamental difference between DBPL and implementations of other persistent programming languages is the fact that DBPL does not rely on a physically or conceptually centralized persistent store. On the contrary, the DBPL storage manager tries to exploit the notion of separate or nested name spaces (modules, relations) and to take advantage of the static distinction between persistent and volatile, recoverable and non-recoverable, shared and non-shared scopes.

As a first consequence, the layer SMS does not map all persistent data objects to a single huge database file (as it is done in the PS-algol, Napier, Galileo or O_2 implementations) but maps top-level database variables of relation types to individual files of the operating system. Furthermore, it maps all database variables of other types (like integers, arrays or records) that are declared within the same database module to a single file that also contains all meta information pertaining to that module and which therefore acts as a kind of distributed database dictionary. At a next higher level of granularity, database modules are mapped to operating system directories. Non-persistent bulk data variables are also stored in individual address spaces that are in general mapped to main memory by the buffer manager and are only paged out to files if the system runs out of main memory resources.

Despite the increased complexity of the low-level storage management, this modularization at the file and directory level significantly increases the usability of the DBPL system in today's operating system environments since it allows dynamic access control for different users and user groups, incremental database backups and reorganizations, distribution across multiple volumes and even the use of distributed file systems to store database variables on different network notes.

The lower levels of the DBPL system (SMS and CRDS) identify atomic fragments (i.e. records without relation-valued subcomponents) by means of tuple identifiers (TIDs) consisting of a page identifier and a record number within the page. Subrecords of arbitrary deeply nested relations are identified by a fixed-sized identifier called super-TID. This super-TID consists of TID for a map structure that is allocated for each complex relation element and a relative offset within that map, identifying a map entry that finally points to the atomic fragment (see Figure 2). The sizes of the three components of a super-TID allow the identification of 2^{32} pages with up to 2^8 objects each. The total number of nested elements of a top-level relation element may not exceed 2^{32}.

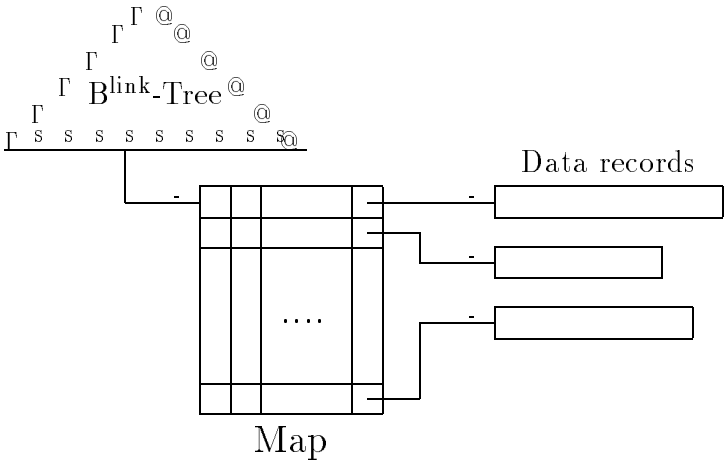

Figure 2: Access path management for nested name spaces

These implementation-oriented names (super-TIDs) are only made visible to the query evaluator to give support for direct-access join algorithms. All other operations of the CRDS either work on full relations or on single relation elements identified by their key-value. The layer CRDS enforces copy semantics for updates (e.g. it performs a deep-copy of a relation element prior to relation insertion). DBPL therefore does not make use of the possibility to implement shared or circular data structures using super-TIDs.

The built-in key selectors (see Section 3.1) for relation variables are implemented by means of B-link trees (extended B-Trees allowing for concurrent updates [LY81]) that map from (composite) key values to TIDs. Hierarchical and key-based access to NF^2-relations is implemented through map structures that are stored for each complex root relation element and that describe the hierarchic relationships between the atomic fragments of all nested relation elements. Each map entry is augmented by a fixed-sized (8 Byte) key fragment to speed up key-based element selection. By separating data records from access structures (maps) and by choosing appropriate chaining techniques within the map, a high degree of parallelism can be achieved for update and lookup operations.

The CRDS level furthermore supports secondary index structures (generalized B-link trees) that map from (composite) attribute values to sets of TIDs. That means, that in contrast to key-selectors, no uniqueness constraint is enforced. Secondary indices can be created dynamically and are intended to accelerate user-defined naming schemes (e.g. like those defined by the selectors ByColor or RedParts).

4.2 Associative Binding and Query Optimization

A distinguishing property of the predicate-based binding mechanisms introduced in Section 3.2 is the strict separation between the specification of bulk data identification (by a first-order predicate in the language) and its implementation by the system in terms of access structures or explicit iterations. Following the approach of relational database systems, DBPL completely hides the implementation of selected relation access from the programmer. Decisions about access path support, clustering and distribution are made dynamically, outside the scope of the language since such decisions have to be based on the evolving access patterns of a multitude of application programs and on the varying sizes of the bulk data structures involved.

This should be seen in contrast to the prevailing notions of "pointers", "mutable values" or "object identifiers" found in other PPLs and DBPLs that correspond more closely to the implementation concept of storage locations allowing for constant-time direct access. Similarly, user-defined iterators as present in E [Ric89], CLU [LAB+81] or Trellis/Owl [SCB+86] are only a first step towards an abstraction from the strict control flow enforced by imperative programming languages. Since users of an iterator still have to consider the order in which iterators are nested or the point in time when selection predicates are evaluated.

Due to the dynamic nature of conditions which determine the choice of an "optimal" access strategies, the DBPL compiler does not generate directly executable code for selector applications (like Parts[SuppliedFrom

("Hamburg")], see section 3.2) but merely passes an augmented syntax tree of the query expression EACH p IN Parts: SOME s IN Suppliers ((s.name=p.supplier) AND (s.city="Hamburg)) to the DBPL runtime system. The layer PSMS is capable of performing symbolic manipulations on such augmented syntax trees. These transformations are necessary to pass parameters to a selector, to bind variable names occurring within a query expression to a global value or to nest, assign and duplicate selectors and constructors.

A selector application appearing in an expression context, e.g.,

```
SOME p IN Parts[SuppliedFrom("Hamburg")] (p.name = "Chair")
```

can simply be replaced by the query expression of the selector body.

```
SOME p IN
   {EACH p IN Parts: SOME s IN Suppliers ((s.name=p.supplier) AND (s.city="Hamburg))}
      (p.name = "Chair")
```

It should be noted that query expressions are *not* evaluated by repeatedly substituting subexpressions by their value. In place of such a "strict" evaluation order, the DBPL system delays "lazily" the evaluation of such expressions until the final context for the expression is known. The rationale behind this strategy is that larger expressions give the query optimizer more flexibility to choose an advantageous execution order for subexpressions and to exploit "semantic" knowledge derived from the full expression context[2]. For example, the DBPL query optimizer (within the layer CPMS) simplifies the above boolean expression to the following expression which only requires two primary key accesses.

```
Suppliers[ Parts["Chair"].supplier ].city = "Hamburg"
```

The DBPL query optimizer normalizes and simplifies a query expression given by an augmented syntax tree by eliminating negations as well as redundant and constant subexpressions. In a next step, standard algebraic optimizations are performed (pushing selections and projections, range nesting for decoupled subqueries [JK83]). Finally, an "optimal" nesting of quantifiers (SOME, ALL, EACH) is chosen. The latter optimization is equivalent to the determination of the join order in algebra-based optimizers and is guided by the existence of index structures provided by the layer CRDS. The support for direct hierarchic access within NF^2-relations (sometimes called "materialized joins") in the DBPL data model gives rise to additional possibilities for access optimization. The simple cost model of the query optimizer is the estimated number of accesses to persistent variables (i.e. the size of intermediate results is not taken into account).

The output of the query optimizer is (conceptually) still a (nested) DBPL expression that becomes subject to "strict" evaluation. The syntax tree for the expression is adorned with attributes that determine which subexpressions are to be evaluated by which evaluation algorithm of the layer CPMS. The current implementation for nested relations provides three evaluation strategies (nested loop, index-based joins, hierarchic access); a previous implementation of DBPL for flat relations also supported grid files for multi-dimensional range queries and semi-join programs. The evaluation either yields a boolean result (for existentially or universally quantified expressions) or a temporary relation. In many cases (e.g. in the assignment Parts:+ rex where Parts does not appear in rex) the selected relation elements of rex do not have to be stored in a temporary relation but can be "pipelined" directly to another part of the system that processes them (e.g. performs element insertions).

The previous descriptions illustrate that the use of selectors in an expression context is roughly equivalent to the concept of "query modification" employed in relational database systems. The basic idea for the implementation of a selected relation *update* R[sp(...)]:= rex is again to pass a symbolic description of sp and rex to the DBPL runtime system that transforms the selected update conceptually into a guarded non-selected relation update (see Section 3.2). The specific structure of the guarded update may allow significant simplifications of the integrity checking code based on knowledge about the structure of rex. For example, if it can be inferred that each element in rex fullfils the selection completely.

To summarize, DBPL delegates the implementation of user-defined naming schemes to the query optimization component of the runtime system that makes heavy use of

- the well-understood semantics of first-order predicates;
- the advantageous algebraic properties of relations as bulk data type constructors;

[2]This "lazy" expansion of nested predicates is also essential for the optimization of recursive constructor applications.

- the strict separation between side-effect free query expressions and arbitrary DBPL procedures and functions enforced in DBPL.

4.3 Extended Naming Schemes and Transaction Management

As mentioned in Section 2.3, DBPL guarantees that concurrent transactions are executed in a serializable schedule. The underlying concurrency control theory [BHG87] assumes data objects to be abstract *named* entities that can be used as arguments in read or write operations. By using different naming schemes one arrives at different definitions of a conflict which finally lead to concurrency control protocols that allow substantially different degrees of parallelism. For example, multi-level concurrency control protocols [BSW88, EM82] utilize fine-grained naming schemes at the lower system levels and more application-oriented naming schemes at the higher system levels. The goal is to reduce the number of conflicts at the application-level by abstracting from details of lower-level naming schemes.

DBPL utilizes a three-level concurrency control and recovery scheme. Names within the lower layer SMS are page identifiers and file identifiers. The medium layer CRDS makes use of super-TIDS to perform synchronization for data records and access structures. The transaction management at the highest abstraction level of the DBPL data model (within the layer CTMS) identifies components of persistent variables (like SparseMatrix[1][4].val) through a path consisting of

- a system-wide unique database module identifier (essentially a time stamp generated at compile-time),
- an identification of a persistent variable within that module (e.g., SparseMatrix), and
- a (possibly empty) path consisting of key values and attribute identifiers (e.g., 1 , 4, val).

Due to the nested structure of persistent variables in DBPL, this naming scheme leads to a multi-granularity concurrency control protocol at the CTMS level. It is therefore possible to name individual data elements as well as large collections of subcomponents of a composite object. Multi-granularity locking protocols [GLP75] allow both, a high degree of concurrency between transactions that operate at a fine level of granularity, and a small synchronization overhead for transactions that access large portions of the database.

The DBPL transaction manager CTMS resides above the query evaluation component CPMS in order to be able to avoid the well-known "phantom problem" [EGLT76]. Roughly speaking, the "phantom problem" occurs if set-oriented naming schemes are mapped to element-oriented naming schemes and synchronization is performed solely on individual data elements. For example, during the execution of a transaction that contains the access expression

```
EACH p IN Parts: p.color = red
```

it is not enough to lock all stored red elements in Parts, but it is also necessary to avoid the possibility that other concurrent transactions insert new elements in Parts making the selection predicate p.color = red true. Such predicative synchronization schemes for DBPL are discussed in more detail in [BJS86].

5 Concluding Remarks

The set- and predicate-oriented approach of DBPL to bulk data identification and manipulation differs considerably from the predominating identification mechanisms of persistent programming languages based on system-generated "object identifiers". This paper emphasizes the need for application-oriented and modular naming schemes with non-standard binding mechanisms as they are provided by the declarative selector mechanism of DBPL. The paper furthermore links the DBPL approach to more conventional naming and binding mechanisms by describing implementation aspects of DBPL's name space management in a persistent multi-user environment.

The authors expect that next generation database programming languages will support a broader spectrum of identification mechanisms. A challenging goal is to maintain DBPL-like associative access without resorting to an "omniscient" runtime system, i.e. to implement the functionality of the DBPL access path

management, query optimizer and integrity checker in a language with only basic built-in identification and structuring mechanisms. This will require a powerful type system to capture the highly polymorphic nature of the query optimizer, reflection capabilities [SSS90] to reason about the structure of selection expressions and language constructs to achieve iteration abstraction [LG86].

References

[ACC81] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7), July 1981.

[AGO89] A. Albano, G. Ghelli, and R. Orsini. Types for Databases: The Galileo Experience. In *Proc. of the 2nd Workshop on Database Programming Languages, Portland, Oregon*, June 1989.

[AM88] M.P. Atkinson and R. Morrision. Types, Bindings and Parameters in a Persistent Environment. In M.P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, Topics in Information Systems. Springer-Verlag, 1988.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman, editors. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BJS86] S. Böttcher, M. Jarke, and J.W. Schmidt. Adaptive Predicate Managers in Database Systems. In *Proc. of the 12th International Conference on VLDB*, Kyoto, 1986.

[BL84] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[Böt90] S. Böttcher. Improving the Concurrency of Integrity Checks and Write Operations. In *Proc. ICDT 90*, Paris, December 1990.

[Bro89] A.L Brown. Persistent Object Stores. PPRR 71-89, Universities of Glasgow and St Andrews, March 1989.

[BSW88] C. Beeri, H.-J. Schek, and G. Weikum. Multi-Level Transaction Management, Theoretical Art or Practical Need? In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 134–154. Springer-Verlag, 1988.

[But87] M.H. Butler. Storage Reclamation in Object Oriented Database Systems. In *Proc. of SIGMOD Conf., San Francisco*, 1987.

[Car86] L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[Car89] L. Cardelli. Typeful Programming. Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989.

[CDMB90] R. Connor, A. Dearle, R. Morrison, and F. Brown. Existentially Quantified Types as a Database Viewing Mechanism. In *Advances in Database Technology, EDBT '90*, volume 416 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 1990.

[Coh81] J. Cohen. Garbage Collection of Linked Data Structures. *ACM Computing Surveys*, 13(3), September 1981.

[Dat89] C.J. Date. *A Guide to the SQL Standard.* Addison-Wesley, second edition, 1989.

[DCBM89] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88 – A Database Programming Language? In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[DD79] A. Demers and J. Donahue. Revised Report on Russel. TR 79–389, Computer Science Department, Cornell University, 1979.

[Dea89] A. Dearle. Environments: a flexible binding mechanism to support system evolution. In *Proc. HICSS-22, Hawaii*, volume II, pages 46–55, January 1989.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notion of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(11):624–633, November 1976.

[EM82] J. Eliot and B. Moss. Nested Transactions and Reliable Distributed Computing. In *Symposium on Reliability in Distributed Software and Database Systems*, pages 33–39, 1982.

[GLP75] J.N. Gray, R.A. Lorie, and G.R. Putzolu. Granularity of Locks in a Shared Data Base. In *Proc. VLDB Conference*, Boston, Mass., September 1975.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *ACM-SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, May 1987.

[Gra81] J.N. Gray. The Transaction Concept: Virtues and Limitations. In *Proc. 7th VLDB Conference*, pages 144–154, Cannes, France, September 1981.

[Har88] R. Harper. Modules and Persistence in Standard ML. In M.P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, Topics in Information Systems. Springer-Verlag, 1988.

[Hoa68] C.A.R. Hoare. Record Handling. In F. Genuys, editor, *Programming Languages*, pages 291–347. Academic Press, London, 1968.

[JGL+88] W. Johannsen, L. Ge, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. Database Application Support in Open Systems: Language Support and Implementation. In *Proc. IEEE 4th Int. Conf. on Data Engineering*, Los Angeles, USA, February 1988.

[JK83] M. Jarke and J. Koch. Range Nesting: A Fast Method to Evaluate Quantified Queries. In *ACM-SIGMOD International Conference on Management of Data*, pages 196–206, San Jose, May 1983.

[JLRS88] W. Johannsen, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. The DURESS Project: Extending Databases into an Open Systems Architecture. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 616–620. Springer-Verlag, 1988.

[JLS85] M. Jarke, V. Linnemann, and J.W. Schmidt. Data Constructors: On the Integration of Rules and Relations. In *11th Intern. Conference on Very Large Data Bases, Stockholm*, August 1985.

[L+77] B. Liskov et al. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8), August 1977.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[Lan66] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[LY81] P.L. Lehmann and S.B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[MAD87] R. Morrison, M.P. Atkinson, and A. Dearle. Flexible Incremental Bindings in a Persistent Object Store. Persistent Programming Research Report 38, Univ. of St. Andrews, Dept. of Comp. Science, June 1987.

[Mat87] D. Matthews. Static and Dynamic Type Checking. In *Proc. of the Workshop on Database Programming Languages, Roscoff, France*, pages 43–52, September 1987.

[MRS84] M. Mall, M. Reimer, and J.W. Schmidt. Data Selection, Sharing and Access Control in a Relational Scenario. In M.L. Brodie, J.L. Myopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*. Springer-Verlag, 1984.

[MRS89] F. Matthes, A. Rudloff, and J.W. Schmidt. Data- and Rule-Based Database Programming in DBPL. Esprit Project 892 WP/IMP 3.b, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, March 1989.

[MS89] F. Matthes and J.W. Schmidt. The Type System of DBPL. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, pages 255–260, June 1989.

[Ric89] J.E. Richardson. E: A Persistent Systems Implementation Language. Technical Report 868, Computer Sciences Department, University of Wisconsin-Madison, August 1989.

[RM87] L. Rowe and Stonebraker M. The POSTGRES Data Model. In *Proc. 13th VLDB, Brighton*, pages 83–96, September 1987.

[RS81] M. Reimer and J.W. Schmidt. Transaction Procedures with Relational Parameters. Report 45, Institut für Informatik, ETH Zürich, Switzerland, October 1981.

[SBK+88a] J.W. Schmidt, M. Bittner, H. Klein, H. Eckhardt, and F. Matthes. DBPL System: The Prototype and its Architecture. Esprit Project 892 WP/IMP 3.2, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, November 1988.

[SBK+88b] J.W. Schmidt, M. Bittner, H. Klein, H. Eckhardt, and F. Matthes. DBPL System: The Prototype and its Architecture. DBPL Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, November 1988.

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An Introduction to Trellis/Owl. In *Proc. of 1st Int. Conf. on OOPSLA*, pages 9–16, Portland, Oregon, October 1986.

[Sch88] J.W. Schmidt. Semantics of Selective Set Assignment. (technical note), December 1988.

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

[SGLJ89] J.W Schmidt, L. Ge, V. Linnemann, and M. Jarke. Integrated Fact and Rule Management Based on Database Technology. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, Topics in Information Systems. Springer-Verlag, 1989.

[SL85] J.W. Schmidt and V. Linnemann. Higher Level Relational Objects. In *Proc. 4th British National Conference on Databases (BNCOD 4)*. Cambridge University Press, July 1985.

[SSS90] L. Stemple, D. Fegaras, T. Sheard, and A. Socorro. Exceeding the Limits of Polymorphism in Database Programming Languages. In *Advances in Database Technology, EDBT '90*, volume 416 of *Lecture Notes in Computer Science*, pages 269–285. Springer-Verlag, 1990.

[Str67] C. Strachey, editor. *Fundamental concepts in programming languages*. Oxford University Press, Oxford, 1967.

[SWBM89] J.W. Schmidt, I. Wetzel, A. Borgida, and J. Mylopoulos. Database Programming by Formal Refinement of Conceptual Designs. *IEEE – Data Engineering*, September 1989.

[Wir83] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.