The Database Programming Language DBPL
User and System Manual

Florian Matthes
Andreas Rudloff
Joachim W. Schmidt
Kazimierz Subieta

The database programming language DBPL is based on the notion of bulk type and iteration abstraction, supports data persistence and transaction procedures, and has Modula-2 as its algorithmic kernel.

This fist part of this document is intended for users of the DBPL system. It gives an introduction into the DBPL language concepts and illustrates their use in a larger modular database application.

The second part gives some insight into the implementation of the DBPL system. As of today, the DBPL system exists in two fully source code compatible implementations, VAX/VMS DBPL and Sun DBPL. Both are written entierly in Modula-2 and consist of a compilation and a run time environment. The multi-user run time system is highly portable and runs under VAX/VMS 6.1, SunOS 4.1, IBM AIX 3.2 and IBM OS/2. Both DBPL systems are integrated deeply into commercially available software development environments (NSE, SCCS, CMS) and provide optimized transactional multi-user access not only to type-complete DBPL databases but also to external relational databases via a fully transparent SQL gateway.

Status: Revised, July 1992

Arbeitsbereich DBIS
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln Straße 30
D-2000 Hamburg 54
Federal Republic of Germany

This work was supported by the European Commission, Basic Research, Contract # 4092, FIDE.

Contents

1 Language and System Support for Data-Intensive Applications

The paper is intended to be read in conjunction with the *DBPL Rationale and Report* [MS92] and reports on the current status of the DBPL system, the result of a long term research and development project at Hamburg University that was set up around 1985. The paper presents two complementary views on the DBPL system: The first and the second section provide an introduction into the DBPL language concepts for database programming in the small (section 1) and for database application development in the large (section 2). The understanding of this "external" conceptual view of DBPL provides the basis for the more technical description of the "internals" of the DBPL system, in particular its compilation system (section 3) and its multi-user database runtime support (section 4). The paper ends with a discussion of our past experience using the DBPL system and possible future application areas.

As of today, the DBPL system exists in two fully source code compatible implementations, VAX/VMS DBPL and Sun DBPL. Both are written entierly in Modula-2 and consist of a compilation and a run time environment. The multi-user run time system is highly portable and runs under VAX/VMS 6.1, SunOS 4.1, IBM AIX 3.2 and IBM OS/2. Both DBPL systems can be integrated deeply into commercially available software development environments (NSE, SCCS, CMS) and provide optimized transactional multi-user access not only to type-complete DBPL databases but also to external relational databases via a fully transparent SQL gateway.

Figure 1 depicts the overall DBPL system architecture and indicates the interaction between the various parts of the DBPL system. A DBPL application typically consists of a collection of modules and interfaces (represented by dashed boxes in Fig. 1). Some of these modules define *shared* objects like library routines or database variables, others define private, application specific code or data. Individual modules and interfaces are translated by the DBPL compiler and then linked with other compiled DBPL modules or object code developed in other programming languages (C, C++, Modula-3) yielding an executable program with (symbolic, type-safe) references to shared libraries and shared databases.

At runtime, this executable program interacts through the interface module *DBPLRTS* with the various layers of the DBPL run time system (*PSMS, CTMS, SQLGate, CPMS, CRDS, SMS*; see Sec. 5). Database objects are either stored in a DBPL-specific format in files accessed through operating system calls issued by the layer *SMS* or they are held in commercial SQL databases (Ingres, Oracle) and accessed via set-oriented dynamic SQL+C statements following the *X-Open* standard for SQL database access. The distinction between DBPL and SQL database objects is fully transparent to the application programmer.

Fig. 1 also outlines the interaction between two DBPL applications (possibly running on different nodes of a TCP/IP or a DECnet local area network) that import a common set of database modules and therefore run against shared databases. The sharing of DBPL database objects is achieved by a page-oriented client-server architecture using a three-level (*CTMS, CRDS, SMS*) concurrency-control and recovery protocol based on explicit message passing. The layer *LMS* provides communication services and a centralized scheduling process that guarantees serializable transaction execution for DBPL applications. Concurrent and possibly remote access to SQL databases is handled using built-in services of the commercial

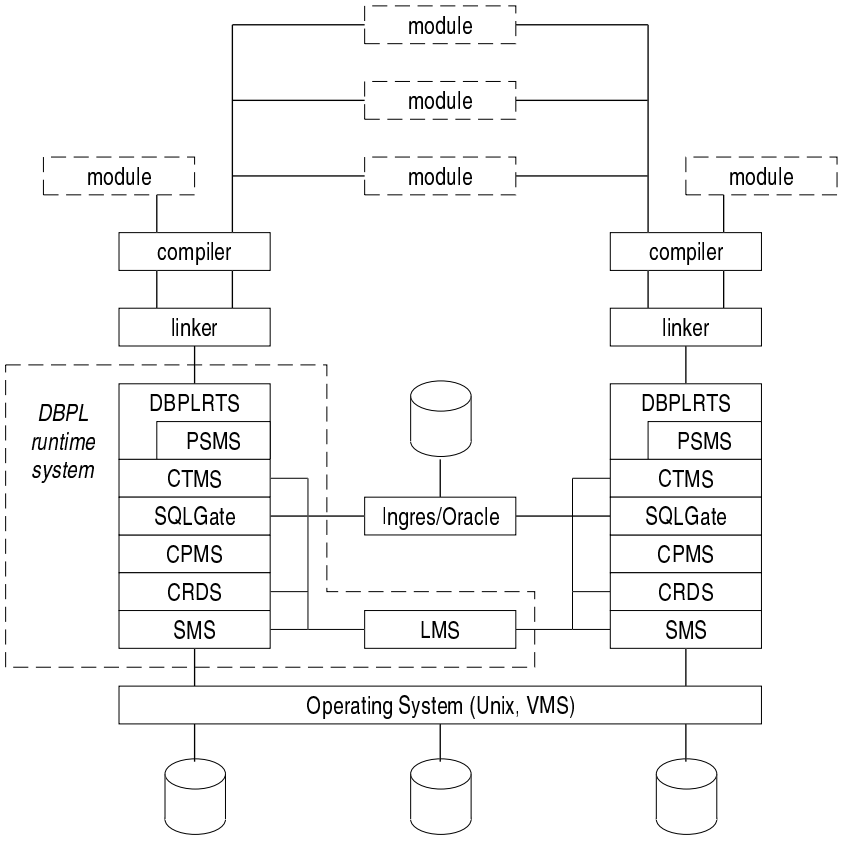


Figure 1: Overall DBPL system architecture

SQL servers.[1]

The next section introduces the notions of bulk type, iteration abstraction, data persistence and transaction procedures as found in DBPL. These concepts are employed in section 2 to solve a larger programming task, the development of a cross-reference database for module, identifier and type dependency management.

[1]There is no two-phase commit protocol to guarantee abort / commit consistency across mixed DBPL and SQL databases in case of system failures at commit time.

2 Language Concepts for Database Programming

The following sections highlight the central database language concepts of DBPL. Programmers familiar with a "traditional" programming language like Modula-2 only need to understand these few additional features to successfully utilize the database capabilities of DBPL. Due to its orthogonal language design, there are no *special* rules governing the naming, typing, binding, scoping, sequencing etc. of these Modula-2 extensions. In particular, it should be noted that every correct Modula-2 program is also a correct DBPL program, i.e., DBPL is a fully upward compatible Modula-2 extension.

The running example used in this chapter is the representation of a directed graph by means of a node and an edge set. A more realistic, self-contained database example is given in the next section.

2.1 Data Type Relation

A relation type declaration specifies a structure consisting of a set of elements of the same type. The cardinality of this set is basically unlimited. Relations extend the notion of sets by allowing to specify a key, i.e. a substructure of the set element that is guaranteed to have a unique value within a relation and, therefore, provides (associative) identification.

In DBPL, the definition of a relation type may contain any nesting of records, arrays, variant records and relations. The only restriction is that on each level of nesting there has to be at least one non-relational attribute and that key attributes of non-first normal-form relations (NF^2 relations) are non-relational.

```
TYPE Nodes  = RELATION OF CARDINAL;
     Edges  = RELATION OF RECORD a, b: CARDINAL; END;
     NF2Rel = RELATION a, b OF
                RECORD
                  a, b: CARDINAL;
                  subrel: Edges;
                END;
VAR  N, NWE: Nodes;
     E: Edges;
     P: NF2Rel;
```

In the example above, `Nodes` and `Edges` are true set types, whereas `NF2Rel` is a nested relation type with key attributes `a` and `b`.

Relation types are "orthogonal" type constructors, i.e. they can occur as elements of arbitrary structures like arrays or records,

```
VAR Table: ARRAY[0..10] OF Nodes;
```

and relation variables can appear in arbitrary scopes and extents, for example, in objects of other data types, i.e.

- as persistent, global, local, dynamic variables, and
- as value and reference parameters.

```
PROCEDURE SwapNodes (VAR n1, n2: Nodes);
  VAR n: Nodes;
BEGIN
  n:= n1; n1;= n2; n2:= n;
END SwapNodes;
```

Note that relations in the "classical" Relational Data Model are simply relations with elements of "flat" record types, i.e. records that only contain fields of the base types (INTEGER, REAL, ...).

Modula-2 identifies elements of arrays by index expressions and record components by field names. Analogically, relation elements are identified by their key values. Therefore, substructures of hierarchical objects are uniformly accessed by a sequence of the previously mentioned identification mechanisms.

```
P[1,5].subrel := Edges{}
```

This assignment replaces the value of the `subrel` attribute of an element `p` with the key values `p.a=1` and `p.b=5` in the relation `P` by the empty relation of type `Edges`.

2.2 Aggregates

DBPL allows the denotation of structured values by means of aggregates, which are simply an enumeration of (possibly structured) values prefixed by the name of the type to be constructed.

```
N:= Nodes{1, 2, 3, 4};
E:= Edges{ {1,2}, {2,3}, {3,4} };
P:= NF2Rel{ {1, 4, E} };
```

As illustrated by the example above, the type identifier can be omitted in nested expressions where the aggregate type can be deduced from its context.

2.3 First-Order Predicates

Any boolean expression in DBPL can contain first-order predicates with existential (`SOME`) and universal (`ALL`) quantifiers ranging over relation expressions. The use of (nested) predicates often allows to avoid explicit iterations over relations and thereby facilitates access optimization to be performed by the DBPL system:

```
  IF SOME n IN N (NOT SOME e IN E (e.a = n)) THEN
    WriteString("There is a node without outgoing edges")
  END;
  IF NOT ALL e IN E (
```

```
    SOME n IN N (e.a = n) AND
    SOME n IN N (e.b = n)) THEN
    WriteString("There are edges from/to non-existent nodes")
  END;
```

First-order predicates play a central role in the formulation of access expressions which are the subject of the next paragraphs.

2.4 Access Expressions

Access Expressions are selection and construction rules for relations. They generalize and unify the concepts of set-oriented retrieval, element-oriented iteration and (updateable) views found in relational database systems.

Selctive access expressions define subrelations of existing relation variables. The following expressions select the elements `e` of the relation variable `E`, fulfiling the selection predicate `e.a=5` respectively `SOME n IN N (e.a =n)`:

```
EACH e IN E: e.a = 5
EACH e IN E: SOME n IN N (e.a = n)
```

The first selection expression selects all edges having 5 as their start node, whereas the second selection expression selects all edges starting from a node contained in the set `N`.

Constructive access expressions denote relation expressions that are derived from combinations of existing relations:

```
{n1, n2} OF EACH n1 IN N, EACH n2 IN N: TRUE
{e1.a, e2.b} OF EACH e1 IN E, e2 IN E: (e1.b = e2.a)
```

The first expression constructs the cartesian product `N`×`N`. The second constructive expression builds a set of edges, containing an edge for each pair of nodes that are connected by a path of length 2.

Please note that access expressions (like boolean expressions) have to be used in a conforming context. There are three possible contexts for access expressions in DBPL:

Relation constructors build a new relation containing (copies of) the elements denoted by an access expression: (set-at-a-time operations)

```
Edges{EACH e IN E: e.a = 5}
Edges{EACH e IN E: SOME n IN N (e.a = n)}
Edges{{n1, n2} OF EACH n1 IN N, EACH n2 IN N: TRUE}
```

Relation iterators iterate over a subrelation denoted by a selective access expression: (element-at-a-time operations)

```
FOR EACH e IN E: e.a = 5 DO WriteCard(e.b, 1) END;
FOR EACH e IN E: SOME n IN N (e.a = n) DO WriteCard(e.b, 1) END;
```

Selector and constructor declarations allow to name and parameterize a given access expression (see section 2.8 and 2.9). (Lambda abstraction)

```
SELECTOR StartAt WITH(x: Node): Edges;
BEGIN EACH e IN E: e.a = x END StartAt;

CONSTRUCTOR AllPossibleEdges: Edges;
BEGIN EACH n1 IN N, EACH n2 IN N: TRUE END Edges;
```

As quantifiers may be nested and the range expression of the inner quantifier may depend on the quantified variable of the outer scope, it is possible to "descend" into nested relations:

```
E:= Edges{e OF EACH p IN P, EACH e IN p.subrel: TRUE}
```

On the other hand, the nesting of aggregates and relational constructors in the projection list of a construction predicate facilitates the construction of nested relations.

```
NF2Rel{ {e.a, e.b, {e} } OF EACH e IN E: TRUE}
```

To summarize, access expressions and first-order predicates form a relationally complete query formalism, exceeding the power of existing relational DBMS.

2.5 Relation Operations

In addition to relation iteration and relation assignment as illustrated above, DBPL has predefined relation operations for set-oriented insertion, deletion and update:

```
E:= E2;  E:+ E2;  E:- E2;  E:& E2
```

Furthermore, there are the usual infix operators (`=, <, >, <=, >=, <>`) to compare two relations of the same type on equality, (strict) set containment, or inequality. Therefore, one can write more succintly `E <= E2` instead of

```
ALL e IN E SOME e2 IN E2 ((e.a=e2.a) AND (e.b=e2.b))
```

2.6 Databases and Persistence

Any module (separate compilation unit) can be declared as a database module by prefixing it with the keyword `DATABASE`. All variables declared in a database module are persistent, i.e. their lifetime is not restricted to a single program execution and exceeds the lifetime of all programs importing them. Furthermore, they are *shared* variables, i.e. they can be accessed by several programs (concurrently). Procedure variables and pointers are not allowed within a database module. The declaration of databases is not restricted to definition modules, which permits the definition of persistent abstract data types.

Persistent variables may only be accessed during transaction execution (see section 2.7). The DBPL runtime system will detect all violations to this rule.

The initialization of persistent variables has to be accomplished before they are accessed for the first time. This is done by extending the original semantics of the module initialization code of Modula-2. Every database module may contain a supplementary initialization code, which is executed only once at the beginning of the existence of a persistent variable.

```
DATABASE MODULE PersistentGraph;
TYPE Nodes = RELATION OF CARDINAL;
     Edges = RELATION OF RECORD a,b: CARDINAL; END;
VAR  N : Nodes;
     E : Edges;

TRANSACTION InitDB;
(* this transaction will be called only one during DB lifetime *)
BEGIN
  N:= Nodes{1};
  E:= Edges{{1,1}};
END InitDB;

DATABASE
  InitDB;
BEGIN
  (* standard Modula-2 initialization code *)
END PersistentGraph.
```

2.7 Transactions

As the unit of concurrency control and recovery, a transaction comprises a sequence of actions and is regarded atomic concerning its effect on the database. Transaction in DBPL can be named and parameterized. In case of nested or recursive calls of transactions, the inner calls are treated as ordinary procedure calls, i.e. DBPL only provides a "flat" transaction model.

The following transactions deletes all nodes in `N` that are reachable through a path of arbitrary length of edges in `E` from a giving starting node `n`. It uses a standard depth-first search traversal strategy.

```
TRANSACTION DeleteSuccessors(n: Node);   (* iterative solution *)
VAR visited: Nodes;

  PROCEDURE VisitNode(n: Node);
  BEGIN
    visited:+ Nodes{n};
    FOR EACH succ IN N:
      SOME e IN E ((e.a = n) AND (e.b = succ)) AND NOT succ IN visited DO
        VisitNode(succ);
    END;
  END VisitNode;
```

```
BEGIN
  visited:= Nodes{};
  VisitNode(n);
  N:- visited;
END DeleteSuccessors;
```

If there are other transactions reading or updating the database variables `N` or `E` simultaneously, the DBPL system will execute them together with `DeleteSuccessors` in a serializable schedule.

2.8 Selectors

Selectors are means to describe value-based constraints on relation variables. The application of a selector defines a selected relation variable, which is equivalent to an updatable view in database systems.

The declaration of a selector has the following general structure:

```
SELECTOR sp ON (R : RelType)
            WITH ( formal parameters )
            FOR ( access restrictions )
BEGIN
  selective access expression
END sp;
```

The ON-parameter allows selectors to be bound to a relation type and not only to a specific relation variable.

```
SELECTOR NodesWithEdges ON (X: Nodes);
BEGIN
  EACH n IN X: SOME e IN E ((e.a = n) OR (e.b = n))
END NodesWithEdges;
```

By declaring access restrictions in the signature of a selector (FOR-parameter), it is possible to restrict the operations allowed on relation variables. The access restrictions are connected to the selector type.

```
SELECTOR ReadNodesWithEdges ON (X: Nodes) FOR (=);
BEGIN
  EACH n IN X: SOME e IN E ((e.a = n) OR (e.b = n))
END ReadNodesWithEdges;
```

Within an expression, the value of a selected relation variable is equal to the value of a relation created by a relational query expression, based on the selective access expression of the selector.

```
NWE:= N[NodesWithEdges]
```

Informally, the update semantics for selected variables is defined as follows:

> updates are executed if and only if a new relation value can be constructed, such that the non-selected part of the new relation is equal to the non-selected part of the relation before the update, and the selected part of the new relation is equal to the right-hand-side of the update statement.

With the current selector semantics, any violation to this constraint will cause the update operation not to be executed at all.

```
N[NodesWithEdges]:= Nodes{1,2,5};
```

As node `5` does not belong to any edge in `E`, the assignment above will not be executed.

By virtue of these semantics of updates on selected relation variables, selectors can be used to enforce value-based integrity constraints on relation variables. Using the optional access restrictions in a selector heading one can constrain access to selected relation variables even further, namely to those kinds of operations explicitely listed in the heading of the selector. Access restrictions are statically verified by the compiler:

```
N[ReadNodesWithEdges]:= N;
```

Since the declaration of *ReadNodesWithEdges* restricts the use of the selected relation variable to read operations, the compiler disallows the assignment above.

2.9 Constructors

Constructors are named and paramaterized construction rules for relations. They are used to define derived relations by a list of constructive access expressions. These expressions refer to relation variables or to other derived relations. One can show that due to the substitution semantics of *WITH* parameters, the constructor formalism in DBPL is more expressive than stratified datalog programs [ERMS91].

2.9.1 Declaration

The signature of a constructor defines its name, formal parameters and result type. Similar to selectors, constructors are first-order objects, which are typed according to their signature. The body of a constructor declaration is a list of relational expressions, defining a relation of the result type (note that the concatenation of access expressions via "," defines a *union* operation):

```
CONSTRUCTOR NonDirectedEdges: Edges;
BEGIN
  EACH e IN E: TRUE,
  {e.b, e.a} OF EACH e IN E: TRUE
END NonDirectedEdges;
```

2.9.2 Application

The application of constructors is restricted to relational expressions (i.e. update operations on derived relations are inadmissible). The substitution of actual parameters is executed according to the rules defined for selectors (see 2.8). The value of a constructor application is obtained by an evaluation of the predicates within the constructor body, which may contain further constructors.

```
E:= Edges{NonDirectedEdges}
```

2.9.3 Recursive Constructors

A set of constructor declarations may contain cyclic references. The semantics of the application of such a recursive constructor is defined by the least fixed point of a system of relation-valued functions. The existence and uniqueness of this fixed point is guaranteed for a (syntactically) restricted class of constructors.

```
CONSTRUCTOR Closure ON (X: Edges): Edges;
BEGIN
  EACH e IN X: TRUE,
  {e1.a, e2.b} OF EACH e1 IN X, EACH e2 IN Edges{Closure(X)}: e1.b = e2.a
END Closure;
```

The above constructor defines a relation that contains an edge for all node pairs that are connected by a (directed) path of arbitrary length.

Using recursive constructors one can express the transaction of section 2.7 that deletes all transitive successors of a given node more declaratively as

```
TRANSACTION DeleteSuccessors(n: Node);
BEGIN
  N:- Nodes{EACH succ IN N:
              SOME e IN Edges{Closure(E)} ((e.a=n) AND (e.b=succ))}
END DeleteSuccessors;
```

This version of the transaction should be as least as efficient as the previous iterative solution and it leaves even more possibilities for access optimizations to be performed by the DBPL run-time system.

3 A Complete DBPL Programming Example

DBPL inherits from Modula-2 linguistic support for the modular development of complex software systems that meet todays engineering demands. This section demonstrates the use of DBPL for the implementation of a larger application and illustrates a systematic approach to database application modularization that turns out to be advantageous also for a wider range of database programs.

The module concept of DBPL is also a key to the type-safe integration of external databases and external libraries developed using other programming languages into DBPL applications, two features that are also demonstrated by examples in this section.

3.1 DBPLXref: A Database for Module, Name and Type Dependency Management

The task of the example application *DBPLXref* is to maintain an online module cross reference database that stores information about

- compiled DBPL module definitions, module implementations and application programs;
- identifiers declared in module definitions (transactions, procedures, types, database variables, etc.);
- module import dependencies and
- identifier dependencies (e.g., variables depend on their types, procedures depend on their parameter types).

This task is accomplished by a small extension to the DBPL compiler to store all identifier information extracted during declaration and body analysis in a DBPL database. The raw data stored in this system-wide database is made available to interactive users by a form-based interface that supports associative and navigational access. Finally, there is a small utility program to remove all identifier information from the database.

The following screen dumps give an idea of the externally visible functionality of the interactive part of the application.

After the compiled and linked DBPL application *dbxref* has been started from a shell, an initial empty entry form for identifier descriptors appears on the screen (see Fig. 2). Question marks denote "don't care" values and all other field descriptors describe a conjunctive query (e.g., "display all type identifiers in file *DBXRefType.def*"). If the user confirms his input, a scrollable list of all matching identifiers appears in a new overlapping window (see leftmost window in Fig. 3).

The user can then browse through the list, return to the initial form to re-specify the query, or pick an identifier. If an identifier is selected, additional information is displayed in a newly created form and a pop-up scroll-menu appears on the screen (see Fig. 3). In the example given, details of a type identifier named *IdentRelT*, declared on line 49 in module *DBXRefType* are displayed. The options of the scroll-menu depend on the kind of an identifier. The possible options for a type identifier include the display of all type identifers on which it depend, the

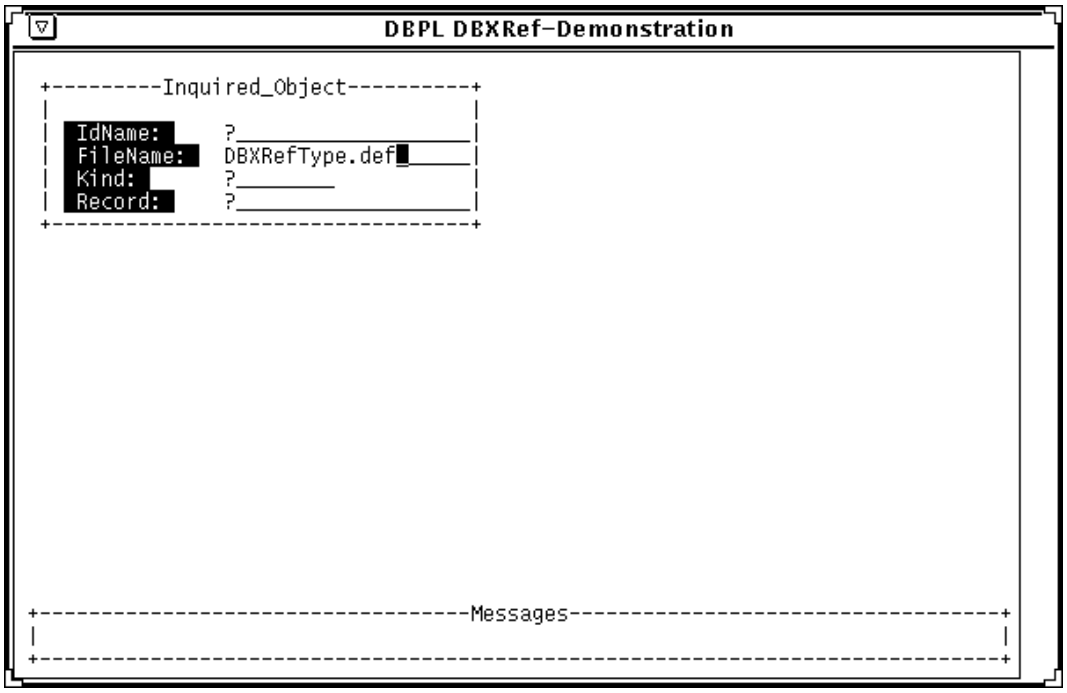


Figure 2: Associative access to identifiers based on their attributes

display of all its occurences in definition or implementation modules, and the display of the source text of its declaration.

Fig. 4 shows the output generated for the selection of the menu item "Used in Implementation Module". Again, the user can scroll through the list of occurences and select a particular item. The selction of an occurrence opens a new window that displays details of the occurence. Another scroll-menu allows the user, for example, to display the context of the occurence or to issue a new associative query in a separate form.

The general idea of the program is therefore to provide easy navigational aids to analyse call, type, and binding dependencies in large DBPL programs composed of a multitude of modules. A particulary useful feature is the stack of past query results displayed as a stack of scroll lists that can be re-activated by an "exit" option available in every menu.

Some care has to be taken in the implementation of the *DBPLXref* application to uniquely identify modules in the file systems of multiple workstations (avoiding name clashes between identical module names) and to control the interaction between concurrent readers (interactive users) and writers (compilation processes).

Fig. 5 depicts the final architecture of the DBPL application program and all its import relationships. The appplication is divided into seven components that define

- a global DBPL database: *DBXRef*, *DBXRefType*
- a procedural interface to the DBPL compiler: *DBXRefO*

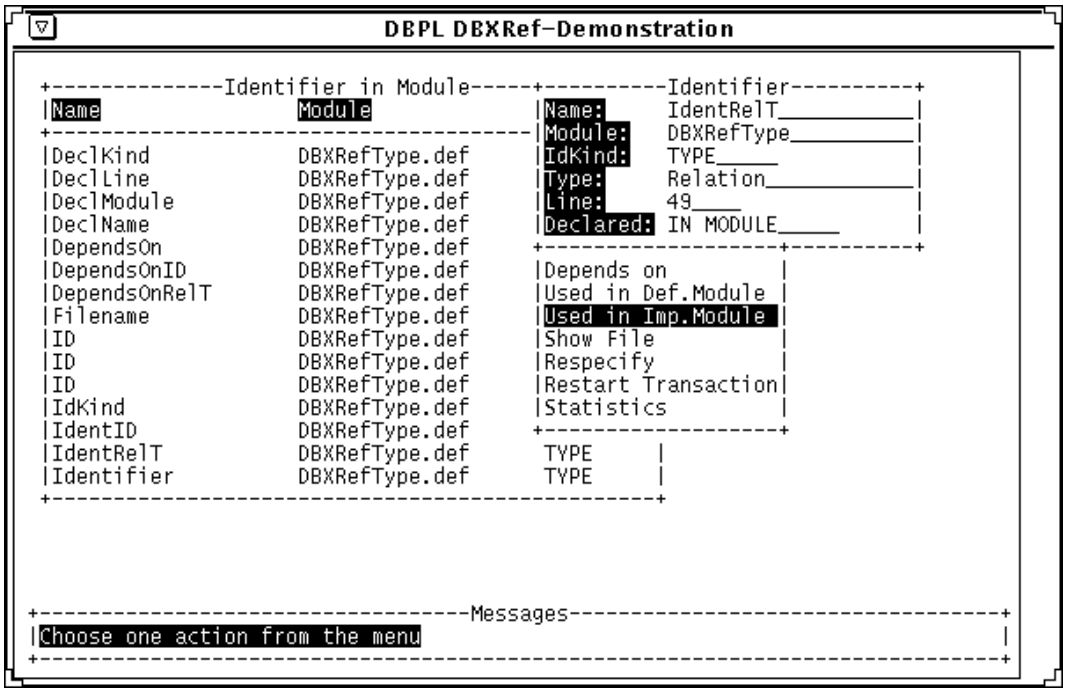


Figure 3: Displaying details about an identifier chosen from a list

- a menu-driven text window interface for browsing: *DBXRefMenu*
- two stand-alone main programs: *dbplxref*, *dbplxrefclean*
- the DBPL compiler itself: *dbplc* plus many additional modules

Shaded boxes in Fig. 5 correspond to implementation modules and main programs whereas the module interfaces (called definition modules in DBPL) are represented as white boxes.

The interested reader can find the full source code of the DBPL application outlined in the following sections as part of the Sun and VAX DBPL distribution.

3.2 Global Type Declarations

Type and constant declarations shared between all parts of the application are factored-out into the module *DBXRefType* that has an empty implementation.

```
DEFINITION MODULE DBXRefType;

CONST
  Maxstring     = 20;
  LongMaxstring = 256;
  UnknownID     = 0;  (* reserved ID value *)
  NotFoundID    = 1;  (* reserved ID value *)
```

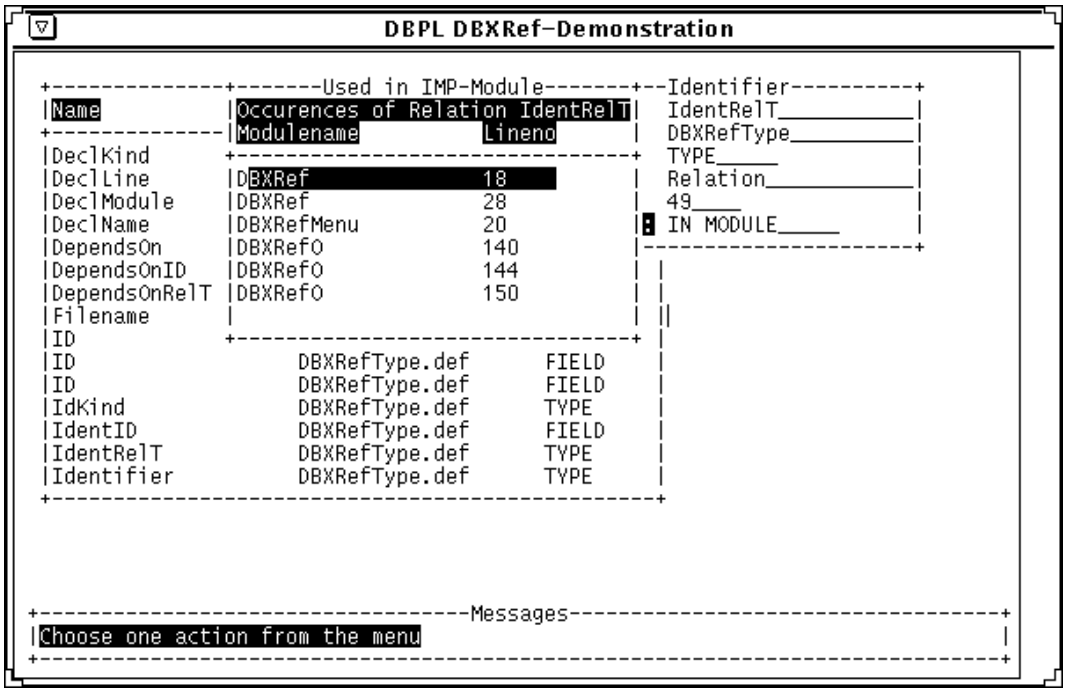


Figure 4: Listing the occurences of a type identifier in implementation modules

```
TYPE
  ID            = CARDINAL;
  TypeClass     = (basesc, setsc, pointersc, relationsc, procsc, enumsc,
                   subrangesc, arraysc, recordsc, predicatesc, dummyc);
  IdKind        = (modulesk, procsk, typesk, constsk, varsk, fieldsk,
                   predicatesk, dummyk);
  ModuleKind    = (defM, implM, mainM, standardM);
  DeclKind      = (inmodule, inrecord, inenum);
  ShortString   = ARRAY[0..Maxstring-1] OF CHAR;
  LongString    = ARRAY[0..LongMaxstring-1] OF CHAR;

  Identifier    = RECORD
    ID          : ID;          (* system-generated unique key *)
    Name        : ShortString; (* identifier declared in a program or 'Anonymous' *)
    Anonymus    : BOOLEAN;     (* is this an anonymous identifier? *)
    Kind        : IdKind;      (* kind of this identifier *)
    Type        : TypeClass;   (* type if kind = typesk else dummyc *)
    DeclLine    : CARDINAL;    (* line number where identifier is declared *)
    DeclModule  : ID;          (* compilation unit where identifier is declared *)
    DeclKind    : DeclKind;    (* the kind of scope for this identifier *)
    DeclName    : ID;          (* the name of the scope for this identifier *)
  END;
  IdentRelT     = RELATION ID OF Identifier;
     (* One entry per identifier [module, variable, type, constant, type,
```

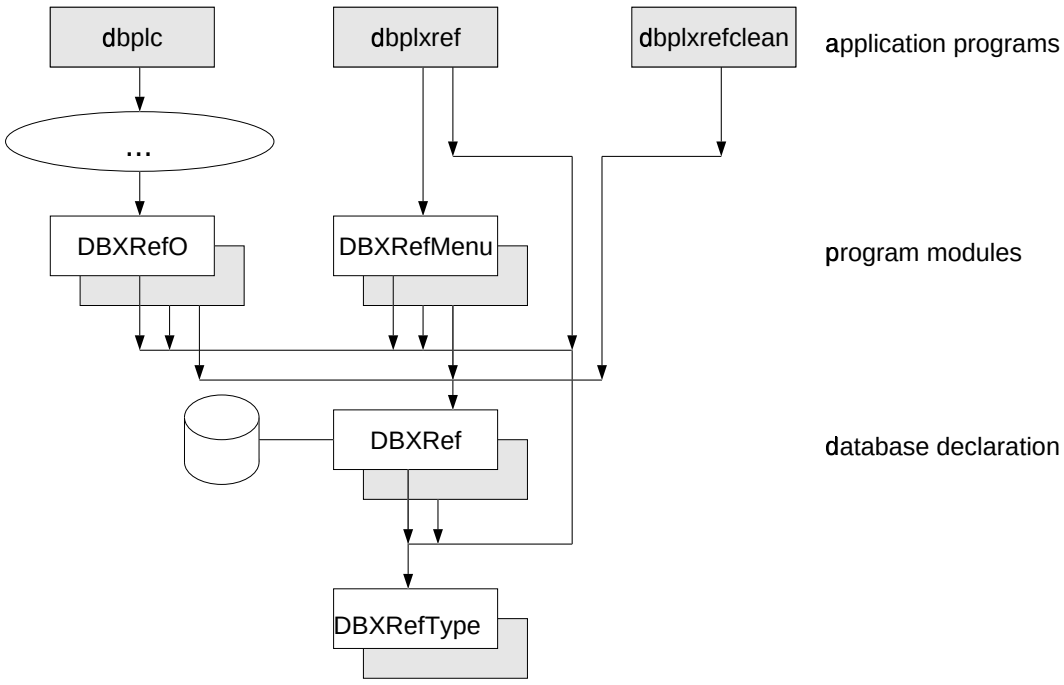


Figure 5: The module structure of the DBPL example application

```
      selector ... record field] declared in a definition module. *)

Module        = RECORD
  ModuleID    : ID;
  Filename    : ShortString;
  MKind       : ModuleKind;
END;
ModuleRelT    = RELATION ModuleID OF Module;
   (* One entry per separate compilation unit. *)

DependsOn     = RECORD
  IdentID     : ID;
  DependsOnID : ID;
END;
DependsOnRelT = RELATION IdentID, DependsOnID OF DependsOn;
   (* Direct, i.e. non-transitive dependency:
      - A module depends on the modules it imports
      - A base type declaration depends on its base type
      - A procedure, transaction, selector or constructor depends on
        its parameter and result types
      - A variable depends on its type
      - ... *)

 Occurence    = RECORD
   ID         : ID;       (* referenced non-module identifier *)
   ModuleID   : ID;       (* used in this module *)
```

```
    Line       : CARDINAL; (* line number in ModuleID where ID is used *)
  END;
  OccurRelT    = RELATION ID, ModuleID, Line OF Occurence;
    (* One entry per utilization(!) of a non-module identifier in a
       definition module. The declaration point is only stored
       in IdentifierRel. *)
END DBXRefType.
```

This example demonstrates the preciseness achievable for database domain definitions in DBPL by the heavy use of enumeration and subrange types. The module essentially exports four relation types (*IdentifierRelT*, *ModuleRelT*, *DependsOnRelT*, *OccurenceRelT*) with their associated element types that are used by clients of this module for record constructors and argument type definitions.

The type *ID* denotes internal, system-generated unique identifiers associated with every identifier declaration that is inserted by the compiler into the database. For example, there may be many declarations of a variable *i*, but each occurence of a variable *i* is assigned the unique *ID* value of its matching declaration. Modules are also uniquely identified by the *ID* of their module identifier. Given the absolute file name of a module or the *ID* of an enclosing scope (module, record, enumeration type) it is possible to determine the *ID* of any DBPL identifier.

3.3 Persistent Variable Declarations

The declaration of the database variables is localized in a single module, *DBXRef*, that exports relation variables and a single *CARDINAL* variable that holds the last identifier value issued. Furthermore, there is a transaction *InitDB* that is automatically called during the initialization of the persistent module but that can be also used to reset the database state at arbitrary points in time.

```
DATABASE DEFINITION MODULE DBXRef;
IMPORT DBXRefType;

VAR  (* persistent database variables *)
  IdentRel        : DBXRefType.IdentRelT;
  ModuleRel       : DBXRefType.ModuleRelT;
  DependsOnRel    : DBXRefType.DependsOnRelT;
  DefModOccurRel,
  ImpModOccurRel  : DBXRefType.OccurRelT;
  IdentCt         : DBXRefType.ID;   (* highest ID key issued up to now. *)

  TRANSACTION InitDB;

END DBXRef.
```

In general, it is a bad idea to allow access to database variables by arbitrary clients since they could violate the database integrity. Most of the integrity constraints of the example can be expressed by quantified DBPL expressions:

```
  ALL i IN IdentRel
     (SOME m IN ModuleRel (i.DeclModule = m.ModuleID) AND
      SOME n IN IdentRel (i.DeclName = n.ID))
```

```
  ALL m IN ModuleRel
     (SOME i IN IdentRel (m.ModuleID = i.ID) AND (i.kind = modulesk))

  ALL d IN DependsOnRel
     (SOME id IN IdentRel (d.IdentID = id.ID) AND
     (SOME dep IN IdentRel (d.DependsOnID = dep.ID))

  ALL o IN ImpModOccurRel
     (SOME id IN IdentRel ((o.ID = id.ID) AND
                           NOT SOME m IN ModuleRel (id.ID = m.ModuleID)) AND
     (SOME mod IN ModuleRel (o.ModuleID = mod.ModuleID))

  ALL o IN DefModOccurRel
     (SOME id IN IdentRel ((o.ID = id.ID) AND
                           NOT SOME m IN ModuleRel (id.ID = m.ModuleID)) AND
     (SOME mod IN ModuleRel (o.ModuleID = mod.ModuleID))

  ALL i IN IdentRel (i.ID <= IdentCt)
```

As discussed in [SM91a, SM90], it is possible in DBPL to only export restricted *views* on database variables that automatically enforce such predicatively specified constraints. Alternatively, one can export read-only views and perform all database updates through verified transactions [SWBM89].

3.4 Encapsulation via Transactions and Procedures

Modifications to the database variables declared in the previous section are limited to the body of seven procedures exported by the module *DBXRefO*. This module and the type module *DBXRefType* are the only parts of the *DBPLXRef* application visible to the compiler:

```
DEFINITION MODULE DBXRefO;
IMPORT DBXRefType;

PROCEDURE InsertIdent(VAR ident      : DBXRefType.Identifier;     (* out *)
                          name       : ARRAY OF CHAR;
                          declModule : DBXRefType.ID;
                          typeClass  : DBXRefType.TypeClass;
                          anonymus   : BOOLEAN;
                          kind       : DBXRefType.IdKind;
                          declLine   : CARDINAL;
                          declName   : DBXRefType.ID;
                          declKind   : DBXRefType.DeclKind);

PROCEDURE InsertModule(VAR module    : DBXRefType.Module;         (* out *)
                           modName   : ARRAY OF CHAR;
                           mKind     : DBXRefType.ModuleKind;
                           filename  : ARRAY OF CHAR);

PROCEDURE InsertDependsOn(VAR dependsOn   : DBXRefType.DependsOn; (* out *)
                              identID     : DBXRefType.ID;
```

```
                                dependsOnID : DBXRefType.ID);

PROCEDURE InsertDefOccurence(VAR occurence: DBXRefType.Occurence; (* out *)
                                identId  : DBXRefType.ID;
                                moduleId : DBXRefType.ID;
                                line     : CARDINAL);

PROCEDURE InsertImpOccurence(VAR occurence: DBXRefType.Occurence; (* out *)
                                identId  : DBXRefType.ID;
                                moduleId : DBXRefType.ID;
                                line     : CARDINAL);

PROCEDURE DeleteDefModules(moduleId: DBXRefType.ID);
(* Delete all information about objects declared in this definition module and
   all (definition or implementation) modules transitively importing
   this module. *)

PROCEDURE DeleteImpModule(moduleId: DBXRefType.ID);
(* Delete all information about occurences and import dependencies of this
   implementation module.
*)
END DBXRefO.
```

The insert transactions provide the necessary object attributes as value parameters and return a newly created identifier via their first variable parameter. This value may then used by the compiler to establish further bindings to an object.

The deletion operations automatically preserve the integrity of the database by removing all transitively dependent declarations. Their implementations make heavy use of the declarative high-level access abstractions provided by DBPL:

```
PROCEDURE DeleteModule(moduleId: DBXRefType.ID);
  VAR lrelident: DBXRefType.IdentRelT;   (* a local relation variable *)
BEGIN
    (* Delete module from module and identifier relation: *)
    ModuleRel:- DBXRefType.ModuleRelT{ ModuleRel[moduleId] };
    IdentRel:- DBXRefType.IdentRelT{ IdentRel[moduleId] };

    (* Delete all dependencies from moduleId to imported objects: *)
    DependsOnRel:- DBXRefType.DependsOnRelT{
                     EACH d IN DependsOnRel: d.IdentID = moduleId};

    (* Determine the set of all locally declared identifiers: *)
    lrelident:= DBXRefType.IdentRelT{EACH i IN IdentRel: i.DeclModule = moduleId};
    (* Remove them: *)
    IdentRel:- lrelident;

    (* Remove dependencies referencing the locally declared identifiers: *)
    DependsOnRel:- DBXRefType.DependsOnRelT{EACH d IN DependsOnRel:
          SOME l IN lrelident((d.IdentID = l.ID) OR (d.DependsOnID = l.ID))};

    (* Remove all occurencies within this module: *)
    DefModOccurRel:- DBXRefType.OccurRelT{
```

```
          EACH o IN DefModOccurRel : o.ModuleID = moduleId};
    ImpModOccurRel:- DBXRefType.OccurRelT{
          EACH o IN ImpModOccurRel : o.ModuleID = moduleId};
END DeleteModule;

PROCEDURE DeleteDefModules(moduleId : DBXRefType.ID);
(* Delete all information about objects declared in this module and
   all modules transitively importing this module. *)
TYPE
  IdRelType = RELATION OF DBXRefType.ID;
VAR
  successors: IdRelType;

  CONSTRUCTOR Successors WITH (root : DBXRefType.ID) : IdRelType;
  BEGIN
    root,
    d.IdentID OF EACH d IN DependsOnRel:
            SOME succ IN IdRelType{Successors(root)} (d.DependsOnID = succ)
  END Successors;

BEGIN (* DeleteDefModules *)
  (* Determine transitive successors of moduleId (incl. moduleId itself) *)
   successors:= IdRelType{Successors(moduleId)};
   FOR EACH succ IN successors : TRUE DO
     DeleteModule(succ);
   END;
END DeleteDefModules;

PROCEDURE DeleteImpModule(moduleId : DBXRefType.ID);
BEGIN
  DeleteModule(moduleId);
END DeleteImpModule;
```

Within the function *DeleteDefModules*, the set of identifiers of modules depending on a definition module is computed by a locally declared recursive constructor *Successors* that is parameterized by the module identifier of the root module.

Since all of these procedures access persistent database variables, they have to be called during a transaction execution (otherwise a runtime error is raised). The compilation of a module X initiates a transaction that first deletes all information transitively referring to a possibly existing "old" version of X and then inserts information about identifiers declared within X. The enclosing transaction guarantees that these operations appear (also for transactions executing in parallel) to be executed as a single atomic action.

3.5 Building Form-Based User Interfaces

A substantial amount of the programming effort in typical interactive database applications is concerned with the management of user-friendly interfaces to support easy, problem-oriented data retrieval. To alleviate this task, the DBPL standard library exports several modules (*Selector, Form, SetForm*) to handle repeating tasks like menu-driven command input, form management and generic display routines for set-structured data.

The following example shows how the menu of Fig. 3 was created:

```
ChooseText := "Depends on|Used in Def.Module|Used in Imp.Module|
              Show File|Respecify|Restart Transaction|Statistics";
s := Selector.Select(level+8,level+41, ChooseText);
CASE s OF
  | 0:  (* Exit key, return to where we came from *)
        status := back;
            EXIT;
  | 1: ... (* Print ``Depends on'' information *)
END;
```

An example for the use of the Form interface is the following procedure that generates the form of Fig. 2 by successive calls to *Form.xxxField* calls, handles user input (*Form.Edit*), and returns the typed input values via variable parameters to the caller. The procedure *Form.EnumField* uses the display string supplied as its third parameter to translate between the internal and external data representation.

```
  PROCEDURE Inquire(VAR i,j,r : ShortString; VAR kind : IdKind;
                        x, y  : CARDINAL;    VAR f    : Form.T) : BOOLEAN;
  VAR z: CARDINAL;
  BEGIN
    Form.Message("Fill in what you know about the object of your interest!", FALSE);
    i:="?";
    j:="?";
    kind := dummyk;
    r:= "?";
    f := Form.Create(x,y,"Inquired_Object");
    Form.ConstField(1, 1," IdName: ");
    Form.TextField(1, 14, i);
    Form.ConstField(2, 1," FileName: ");
    Form.TextField(2, 14, j);
    Form.ConstField(3, 1," Kind: ");
    Form.EnumField(3, 14, "module|proc|type|const|var|field|predicate|?",kind);
    Form.ConstField(4, 1," Record: ");
    Form.TextField(4, 14, r);
    Form.Display(f);
    RETURN (Form.Edit(f));
  END Inquire;
```

Because of the availability of generic set types in DBPL, the management of scroll lists, repeating groups etc. is as simple as the handling of forms for atomic data values. For example, the window of Fig. 4 is created as follows:

```
PROCEDURE CreateModIdRelForm(VAR modIdRel : ModIdRel;
                             VAR modIdRec : ModIdRec;
                                 line,col : CARDINAL): SetForm.T;
(* Return Setform identifier for future display *)
VAR header,body : Form.T;
BEGIN
  header := Form.Create(line,col,"Identifier in Module");
  Form.ConstField(0,0,"Name");
```

```
  Form.ConstField(0,20,"Module");
  Form.ConstField(0,40,"Kind");
  body := Form.Create(line,col,"");
  Form.TextField(0,0,modIdRec.Identname);
  Form.TextField(0,20,modIdRec.Modulename);
  Form.EnumField(0,40, KindText,modIdRec.Kind);
  RETURN SetForm.Create(header,body,modIdRec,modIdRel,15);
END CreateModIdRelForm;
```

The variable *modIdRec* is used as a buffer to make the result of a user selection immediately available for further processing.

The interaction between DBPL and so-called foreign language libraries is exemplified by the module *Subprocess* that is implemented using the Unix library calls *fork* and *wait*. In the example application, this module is used to spawn new editor subprocesses to display DBPL source code.

```
DEFINITION MODULE Subprocess;
TYPE
  ProcessId = INTEGER;
  String = ARRAY[0..79] OF CHAR;

PROCEDURE UnixCall(program: ARRAY OF CHAR;
                   VAR arguments: ARRAY OF String): ProcessId;
(* Call program with these arguments. Each argument and the program
   has to end with a 0C. The program runs as a subprocess.
*)

PROCEDURE Wait(VAR result : INTEGER) : ProcessId;
(* Wait for any subprocess to terminate. Return its process id and
   completion status.
*)
END Subprocess.
```

3.6 Database Access Optimization

The following example gives an idea of the kind of queries that arise in typical DBPL applications. The following query computes the result displayed in Fig. 3, namely the name, source file name, kind and scope identifier for each identifier declared in a file specified by the value of a (local) variable *File*, by taking the the union of a two-way and a three-way join over the global database relations:

```
  GlobalRelT{{i.Name, m.Filename, i.Kind, empty, i.ID} OF
                EACH i IN IdentRel, EACH m IN ModuleRel:
                    (i.DeclModule = m.ModuleID) AND (m.Filename = File)
                    AND (i.Kind <> fieldsk),

             {i.Name, m.Filename, i.Kind, decl.Name, i.ID} OF
                 EACH i IN IdentRel, EACH m IN ModuleRel, EACH decl IN IdentRel:
                     (i.DeclModule = m.ModuleID) AND (m.Filename = File)
                     AND (i.Kind = fieldsk) AND (decl.ID = i.DeclName) };
```

The current version of the DBPL query optimizer will use indexed access to the relations *ModuleRel* and *IdentRel* via their primary keys and evaluate filters (like *i.Kind* <> *fieldsk*) as early as possible.

Query optimization becomes even more crucial in the presence of recursive queries, e.g. the following fixed-point computation of the set of identifiers on which a given identifier *root* transitively depends on:

```
CONSTRUCTOR NextModule WITH (root : ID) : IdRelT;
BEGIN
  root,
  d.IdentID OF EACH d IN DependsOnRel:
    SOME s IN IdRelT{NextModule(root)} (d.DependsOnID = s)
END NextModule;

modules := IdRelT{NextModule(Id)};
```

3.7 Transparent Access to Commercial SQL Servers

The current implementations of VAX DBPL and Sun DBPL provide gateways to external Ingres databases (see also Sec. 5.4). The following steps have to be taken to store identifier information (i.e. the relation *DBXRef.IdentRel*) in an Ingres database and not in a DBPL database. First, an Ingres table has to be created that provides attribute and domain declarations as required by the DBPL record type *DBXRefType.IdentRec*:

```
create table identrel(
        id i4 not null not default,
        name c20 not null not default,
        anonymus i1 not null not default,
        kind i1 not null not default,
        type i1 not null not default,
        declline i4 not null not default,
        declmodule i4 not null not default,
        declkind i1 not null not default,
        declname c20 not null not default /* should be i4 */)
with noduplicates,location = (ii_database);
```

Next, the primary key constraint (as required by the type *DBXRefType.IdentRelT*) needs to be enforced in Ingres, for example,

```
modify identrel to btree unique on
  id with nonleaffill = 80, leaffill = 70, fillfactor = 80;
```

The binding between the table *identrel* in a SQL database *xref* and the variable *IdentRel* in the module *DBXRef* is established by a form-based DBPL binding tool shown in Fig. 6. This tool also verifies the compatibility between the SQL table structure and the DBPL type structure. Errors (e.g. the mismatch between the string type for the SQL attribute *declname* and the integer type for the DBPL record field *DeeclName*) are reported in a subwindow (see Fig. 6) and have to be removed before a binding can be successfully established.

Further access to the variable *DBXRef.IdentRel* is fully transparently handled by the Ingres database server. Since the coupling between DBPL and Ingres is done at the level of set-oriented queries, effective optimizations can be performed on the server side. Furthermore, DBPL and Ingres objects can be freely mixed in DBPL expressions (see also Sec. 5.4). In particular, parameterized and recursive views can be defined on external SQL relations.

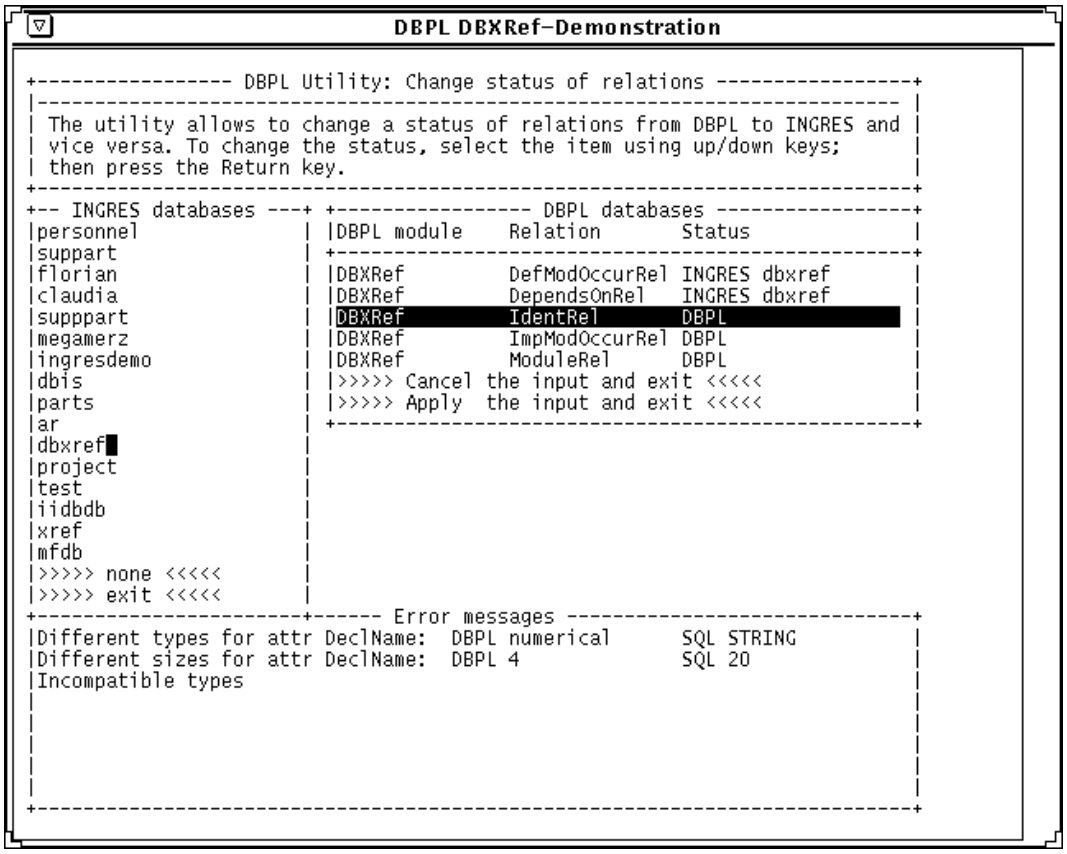


Figure 6: Binding of a DBPL database variable to an external SQL relation

Finally, Fig. 7 shows that external DBPL relations can be accessed using standard SQL statements, e.g. to grant access rights, to define secondary indices and to define parameters that determine the physical database schema.

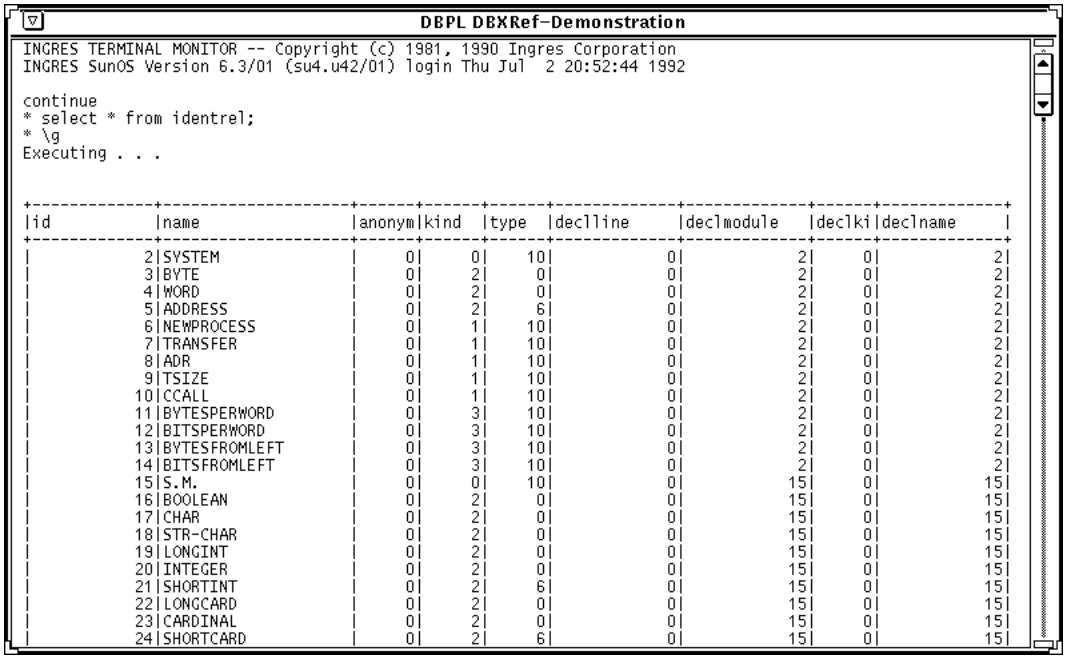


Figure 7: Browsing external DBPL databases using SQL ad-hoc queries

4 The Optimizing DBPL Compiler

Since DBPL is an upward compatible extension of Modula-2, a language processor for DBPL has to address all aspects of state-of-the-art compilation technology, ranging from lexical, syntactic and semantic analysis over error handling to program translation and optimization [ASU86, KMP82, RA83].

In this section, the focus will be on *extensions to programming language technology* to adequately support specific requirements of large scale, long-lived and data-intensive applications. Generally speaking, it turns out that traditional programming language technology provides well-engineered and systematic approaches to local program analysis and standardizable static translation tasks, whereas some specific database system tasks rely heavily on "global", program-wide (or even system-wide) information gathering, possibly during program execution. Therefore, an important task in the design of the DBPL system was the division of labour between the various DBPL system components. Program analysis and code optimization is performed statically by the DBPL compiler, the linker verifies the overall consistency between separately developed system components at application link-time, while the run time system *DBPLRTS* cares for dynamic aspects of database applications like query optimization, storage management, serialization of transactions and failure recovery for persistent data.

4.1 Input to the Compiler

The languages Modula-2 and DBPL support the partition of large programs into independent modules. Each module consists of the definition of an interface and an implementation. Both parts are called *compilation units* as they can be modified and compiled independently. The DBPL compiler checks the consistency between the imports and exports of the individual modules. The compiler accepts also interface descriptions for modules that were implemented in other programming languages (*C*, *FORTRAN*, *Pascal*, *Ada*, ...). Thus, DBPL programs can also be linked with modules of these other languages.

The built-in compiler rules that determine the mapping from DBPL modules to their persistent representations (source code, object code, compiled interface definition, database files, executable files) can be overriden by means of arguments passed to the compiler or environment variables managed by the operating system.

4.2 Output Generated by the Compiler

A *symbol file* results from the compilation of a module interface definition. This file contains, besides of a compact representation of the interface declaration, additional information necessary for type checking between various modules, compatibility tests, and the allocation of variables, procedures, and constants.

The result of the compilation of an implementation module is a *machine program* containing references to data objects and code segments of other modules. If a program includes database operations, the DBPL compiler creates additional, external references to operations of the run time support (see Sec. 5.1). The VAX DBPL compiler directly creates relocatable object code

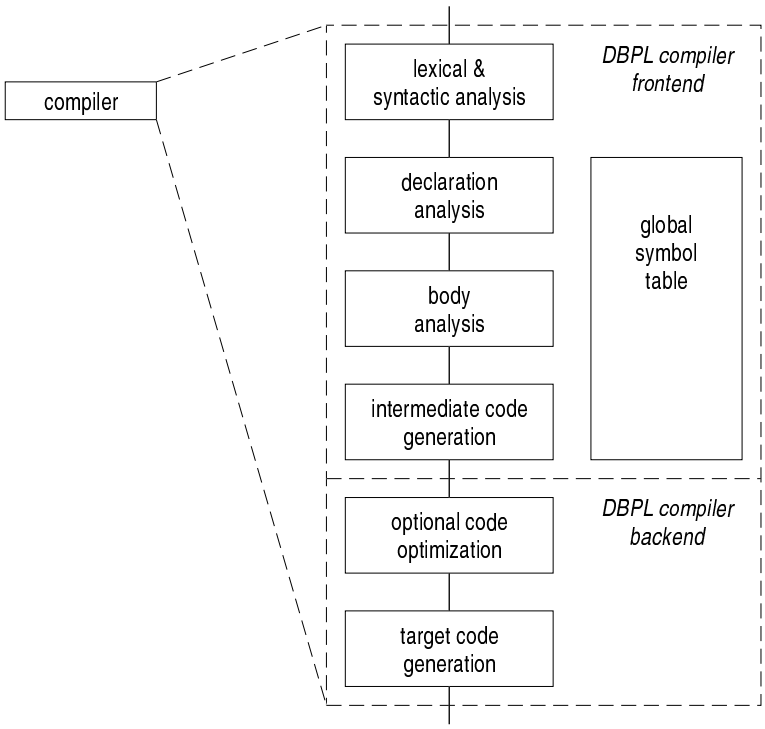


Figure 8: DBPL compiler architecture

while the Sun DBPL compiler generates a portable intermediate representation of abstract machine instructions. This proprietary Sun IR code is shared by all Sun compiler front ends (*FORTRAN*, *C*, *C++*, *Pascal*) and provides a machine-independent common basis for highly optimizing compiler back ends for Motorola, Intel and Sparc target architectures [Muc90].

The VAX and the Sun compiler both extract detailed information about linguistic objects (variables, types, procedures, statements) occuring in a DBPL program. This information is made accessible (in various formats) to other tools in the DBPL environment, for example to support automatic inter-module dependency checking, high-level interactive debugging, effective source code browsing or sophisticated program profiling.

While a database module is being translated, the compiler creates, if necessary, an empty database and an internal description of the database scheme in the *data dictionary* via calls to the DBPL database system (see Sec. 5.1).

4.3 Overall Compiler Structure

The DBPL compiler front end (see Fig. 8) translates a module in four passes. Each pass is being executed by completely independent parts of the compiler. The communication between the passes takes place via main memory data structures containing a compressed representation of all objects (constants, types, variables, and procedure signatures) declared in the program and its system environment, and via sequential interpass files. Thereby, the output of pass i is the input for pass $i+1$. The output of pass 4 is either an object program in the VAX-11 link format, extended by information for the run time debugger or a file in the

Sun IR format. The individual passes have the following functions:

Pass 1: Lexical and syntactical analysis. The source text is partitioned into individual symbols, stored as *tokens* in the interpass file. Identifiers are collected in a symbol table and substituted by unique tokens as well. The output of pass 1 is a syntactically correct DBPL program, augmented by declarations of the imported modules.

Pass 2: Analysis of all declarations within the imported definition modules and within the module that actually has to be compiled. For variables and constants, memory is allocated in the address space of the compiled program. Simultaneously, the compiler creates pointer structures as a one to one image of all declarations within the program, such that type and address information are available in the following passes. Statements are left unchanged and simply passed on to pass 3.

Pass 3: For definition modules, the compilation ends with the output of a symbol file. Otherwise, the actual statements (procedure bodies) are analyzed. This includes the check of type compatibilities within expressions, statements, and procedure parameters.

Pass 4: The DBPL front end does not perform any significant optimizations. Therefore, the code generation can be executed by means of a single scan, one statement at a time. Basically, the input of pass 4 is a copy of the bodies of the different procedures and modules of a program unit.

Back End: The Sun DBPL compiler utilizes Sun's compiler-back end that not only supports code generation for various target architectures but that also applies state-of-the-art code optimizations like tail call optimization, automatic inlining, aggregate breaking, loop-invariant code motion, strength reduction, common subexpression elimination, copy and constant propagation, register allocation, loop unrolling or unused code elimination.

The DBPL front end has to perform a careful program analysis to support the aggressive optimization technologies employed in the back end. The clean separation between the DBPL compiler and its run time support (as described in Sec. 5) turned out to greatly simplify this task, e.g. by shielding the compiler from possible aliasing of cursor variables, sharing of buffer frames or concurrent updates on shared variables.

The following two subsections give some insight into the novel problems arising in the analyisis and translation of a database language and approaches to their solution in the DBPL compiler.

4.4 Program Analysis

Besides of trivial changes to the *scanner* for the recognition of the new keywords (*CONSTRUCTOR, ON* etc.), the *parser*, which is based on the principle of *recursive descent*, had to be extended by procedures to identify the various new productions of DBPL.

Although DBPL has a *LL(1)-grammar* (i.e., can be analyzed with a one symbol lookahead without backtracking), symbol sequences within construction predicates are rearranged in pass 1 in order to simplify the (strictly sequential) code analysis. Since the exertion of these

rearrangements can also be nested (nested relational expressions for NF^2relations), pass 1 was extended by a stack, its elements being lists of symbols.

The internal data structures of the compiler had to be expanded in order to describe the following objects:

Variant Records: Code generation for aggregates and the analysis of relational queries requires more detailed information about the branches of a variant record than present in Modula-2 compilers.

Relations: A relation type representation consists of the relation element type and a fully expanded list of atomic key components.

Selectors, Constructors: In opposition to a procedure, the signature of a selector or a constructor is described by two parameter lists, by a result type and by a set of access restrictions in case of a selector.

Modules: Global modules can have the additional attribute *database*.

Variables: The compiler distinguishes internally between variable parameters, value parameters, ON-parameters, WITH-parameters and "normal variables". The latter are further divided into global variables (static allocation), local variables (automatic allocation), variables at absolute addresses (no allocation), variables in separate compilation units, persistent variables in database modules and variables in quantified boolean expressions and selective access expressions.

In addition to obvious type checking extensions for relation, selector and constructor types, the step from Modula-2 to DBPL introduces the following three qualitatively new program analysis tasks:

1. Loop variables in **for each** statements and quantified expressions have a scope that is local to a statement or a subexpression. The compiler has to resolve bindings for such (overlapping) scopes.

2. In particular cases, the type of an aggregate or of a constructor can only be determined based on information about the context in which it is to be used. This requires a limited form of "target typing", as found, for example, in compilers for the programming language ADA.

3. The possibility to attach access restrictions ("read", "insert", "update", ...) to selected relation variables requires a more detailed *mode* checking than in traditional Modula-2 compilers that essentially distinguish between immutable values and mutable variables only. Compared with the cost of traditional dynamic run-time access control in database management systems (measured in system complexity and execution time), the mode checking extensions to the compiler incurred a negligible overhead.

4.5 Program Translation

Virtually all extensions from Modula-2 to DBPL were undertaken without interference with the machine dependent parts of the code generation in order to obtain a high portability of

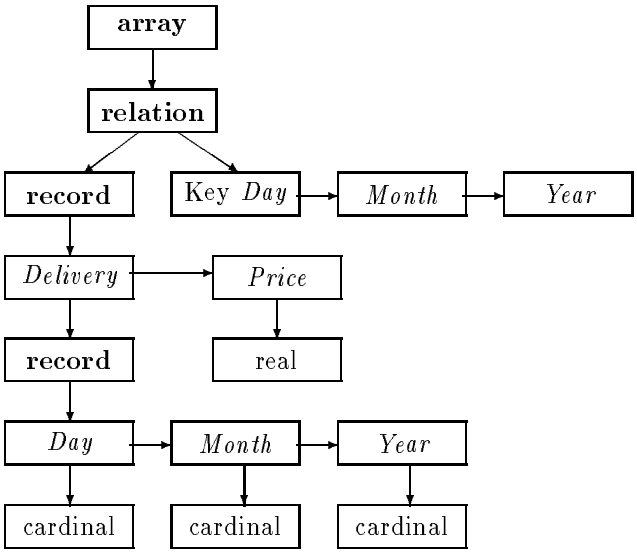


Figure 9: Run-time type descriptions

the compiler. Therefore, many operations of DBPL are (conceptually) implemented by sequences of equivalent statements of Modula-2, containing calls to the runtime support module *DBPLRTS* (see 5.1). Accordingly, the VAX DBPL and the Sun DBPL compiler utilize the same compilation strategies despite significant differences in details of the compilation process (register allocation, code generation).

Each of the following sections deals only with one particular aspect of the compilation of DBPL programs. However, the reader should be aware of the fact that the language principle of *orthogonality* stressed by the DBPL language definition requires implementation strategies that cover arbitrary combinations of these individual compilation patterns.

4.5.1 Aggregates

For the construction of an aggregate, the compiler reserves storage in the local address space of the currently compiled procedure. The elements of a record or an array are stored consecutively in this storage space. Subsequently, the aggregate can be delivered directly to a procedure or stored in a variable. Nested aggregates are created in place.

4.5.2 Run Time Type Descriptions

Many operations of the DBPL run time system are *generic* (i.e., their operands can pertain to different types). For example, the operation *DBPLRTS.CreateRelation* is generic in the sense that it can be used to create relations of parts, relations of suppliers or sets of integers. The actual type is defined by a supplementary type description generated by the compiler. Such a type description has a tree structure as shown in figure 9. Thereby, type constructors are identified by inner nodes, their sons represent the corresponding element types, whereas leaves are formed by the built-in simple types (*CARDINAL*, *INTEGER* etc.).

```
TYPE Date = RECORD Day, Month, Year: CARDINAL END;
  Receipts = RELATION Delivery OF RECORD Delivery: Date; Price: REAL; END;
  Table = ARRAY [1..20] OF Receipts;
```

The generation of such a hierarchical type description is exerted once at runtime in postorder (i.e., starting at the leaves). Each node contains additional information, depending on its type, e.g. the byte size of a simple type value, offset, name and size of a record component, index bounds and element sizes of an array, order of succession and position of the primary key components of a relation.

4.5.3 Persistent Variables

The compiler employs the operations *CreateDB* and *CreateDBVar* of the runtime system (*DBPLRTS*) in order to create a new database during compilation. Thereby, the structure of a database variable is defined by a type description (see previous section).

The compiler inserts code at the beginning of the module initialization to open the databases that belong to an application program. It also generates code to perform the user-defined initialization operations for a database that is opened for the first time. The compatibility between the persistent variables at run time and the database description utilized at compile time is checked by means of a time stamp.

Every access to a non-relational persistent variable is indirect (i.e., the application program utilizes a pointer to the value of the persistent variables). In order to enable a synchronized multi-user access and an efficient buffer management, every access to a database variable via these pointers is enclosed by the operations *GetDBVar* and *ReleaseDBVar* generated by the compiler.

4.5.4 Temporary Variables

Values of relation, selector and constructor types are implemented as instances of *abstact data types* (ADTs). Thus, for objects of these types, the compiler merely allocates pointer variables. The creation, deletion and modification of these objects is accomplished by means of runtime system calls. For example, local relations are created at the beginning and deleted at the end of a procedure, so that relations are allowed within recursive procedures as well. The parameter passing of relations, selectors, and constructors may require the generation of temporary copies of these objects.

4.5.5 Transactions

Transactions are translated by the compiler like procedures, extended by an additional *prologue* and *epilogue*. By means of the prologue, the runtime system is provided with the operation *BeginTransaction* and possibly with supplementary information concerning persistent variables (e.g., database relations) that may be accessed by the transaction in read or write operations. This information gives rise to important optimizations in the DBPL system, e.g. deadlock prevention by preclaiming or special treatment of read-only transactions.

An *EndTransaction* operation is generated in the epilogue of a transaction to initiate a transaction commit. The implementation of DBPL transactions requires a limited form of *exception handling*. During the execution of a transaction body, application-generated exceptions (division by zero, user abort) lead to a controlled abort of the transaction (UNDO). Exceptions generated by the DBPL runtime system provide a mechanism to restart a transaction at arbitrary points in time (e.g. if a deadlock is detected). The compiler generates appropriate information needed by the VMS / Unix operating system to perform the necessary procedure stack unwinding operations for such exceptions.

4.5.6 Identification of Relations

The existence of NF^2relations and selected relation variables requires a uniform and efficient identification mechanism for relation variables at the interface to the runtime system. The chosen mechanism (ADT *Relation* in the layer *DBPLRTS*, see Sec. 5.1) satisfies the following demands:

- The actual operations on relations are separated from the selection of subvariables in hierarchical objects.
- Nested NF^2relations are treated exactly like normalized relations.
- Selected relation variables defined by selector applications can occur as operands in relational operations.

4.5.7 Relation-Valued Operations

DBPL operations on individual relation elements (*GetTuple*, *UpdateTuple* etc.) and navigational operations (*Lowest*, *Next* etc.) are mapped directly to operations of the runtime system. Relation-valued operations (quantified predicates, calculus expressions, assignments of entire relations to relation variables, applications of selectors), however, are not realized by means of (nested) loops in the object code. Instead, the runtime system receives a compact internal representation of the operation(s) in form of a *predicate tree* containing attributes. The execution of an arbitrarily complex relation-valued operation is accomplished in three steps:

1. Evaluation of simple (non-relational) parts of the expression and address computations, utilizing *inline code*.
2. Generation of a predicate tree by means of a sequence of operations of the runtime system, thereby binding operands to program variables.
3. Evaluation of the operations on the dynamically bound operands as defined by the predicate tree.

The predicate tree generated in step 2 (see Fig. 11, p. 42) contains, if necessary, additional references to type structures as in Fig. 9. The generation of a predicate is performed node by node in a preorder traversal.

4.5.8 Selectors and Constructors

Selectors and constructors are represented at runtime exclusively by means of predicate trees. Therefore, predicate trees are allowed to contain parameters as well as (possibly cyclic) references to other predicate trees, together with a list of actual parameters. The runtime system provides symbolic operations for the manipulation of predicate trees (copy, delete, expand, ...; see Sec. 5.2). With the help of these elementary operations, the compiler can implement selector and constructor declarations as well as variables, partial parameter substitution, selector applications and the usage of selectors and constructors in relational constructors and iterators.

4.5.9 Iterators

In DBPL, iterations over relations that are selected by predicates may modify the value of the range relation by assignments to the control variable. Already at the beginning of the iteration, the runtime system receives not only the identification of the range relation variable and the description of the predicate that has to hold for every element of the iteration, but also information about the kind of access that is performed on the iteration loop variable (read and/or write access). Again, this static compile-time information turns out to be extremely valuable for bulk iteration optimizations in the runtime system.

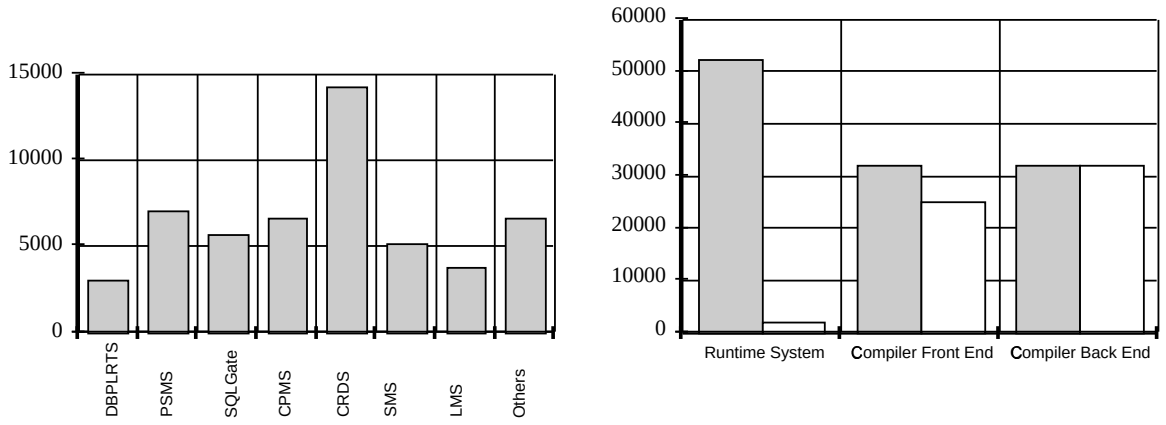


Figure 10: Relative sizes of DBPL system components (lines of Modula-2 code)

5 The Multi-User DBPL Database System

The layered architecture of the DBPL run time system (see Fig. 1 on page 4) is roughly equivalent to the architecture of other relational or object-oriented database systems [SM91b, C+86, PSS+87, Sto90]. It is excelled by its support for non-relational persistent objects, temporary relations, recursive queries, complex objects and client-server architectures. Fig. 10 gives an idea of the relative complexity of the various DBPL system implementation tasks measured in lines of Modula-2 source code. The size of the layer *CRDS* stems from the rather complex interaction between the data access, concurrency control and recovery tasks that arise in the mapping of recursively nested data objects onto fixed-sized data structures. The second diagram compares the complexity of the three major DBPL system components (compiler front end, compiler back end and runtime support) with their counterparts in a typical Modula-2 system implementation.

5.1 DBPLRTS – The DBPL Runtime System Interface

The module *DBPLRTS* represents the only interface to the runtime system. It is accessed by compiled DBPL object programs and by interactive tools (e.g. the DBPL database browser) that require database system functionality. The foremost function of this module is the isolation of applications from implementation details of the database system.

DBPLRTS exports the following abstract data types: values of type *Type* are runtime representations of DBPL type structures (see Fig. 9), *Databases* identify open databases during program execution, *Expressions* identify DBPL query expressions which are evaluated and optimized by an interpreter at run time, *Relations* identify relation variables, and *Transactions* identify active transactions.

The data type *Relation* may illustrate the power and orthogonality that can be achieved by the consequent use of abstract data types in the DBPL system. A *Relation* value at the *DBPLRTS* interface may denote a normalized or a non-normalized relation, a base relation variable or a relational attribute, a persistent database relation or a temporary intermediate result. Furthermore, a *Relation* may be equipped with a *selection predicate* that defines integrity constraints that are to be preserved by relational updates on the relation variable. By virtue of this uniform identification mechanism, a given *DBPLRTS* operation (e.g. *AssignRelation*) which accepts parameters of type *Relation* can be used in a large number of programming situations and can still provide tailored implementations for special parameter combinations (e.g. assignments between temporary relations).

In addition to these types, the interface *DBPLRTS* exports the semi-abstract data types *Bytesize* and *Address* which are needed for the identification of program variables, database variables, relation element buffers and attributes within relation elements.

DBPLRTS provides all relevant functions required for each of the exported ADTs listed above:

- Definition of DBPL type structures (see figure 9);
- Definition of new databases and database variables (*CreateDB*, *CreateDBVar*), enumeration of the variables, indices and types of a database module;
- Opening and closing of databases as well as binding of scalar database variables to program addresses (*OpenDB*, *CloseDB*, *OpenDBVar*);
- Transaction management (*BeginTransaction*, *EndTransaction*, *Commit*, *UseExpression*, *TransactionBody*, *HandleException*);
- Synchronization of the access to non-relational database variables (*GetDBVar*, *ReleaseDBVar*);
- Operations concerning relations and predicates (i.e., all operations provided by the layer *PSMS*, see Sec. 5.2);
- Iteration loops (*BeginIteration*, *Step*, *StopIteration*);
- Synchronized access to relation elements via their primary key value (*GetTuple*, *ReleaseTuple*);
- Creation, deletion and update of hierarchically structured objects (*CreateObject*, *DropObject*, *AssignObject*).

Most of the functions mentioned above are implemented in the lower layers *PSMS*, *CTMS* and *CRDS*. Only the following functionality is realized within the layer *DBPLRTS* itself:

- Exceptions (e.g., division by zero) occuring within an application are handled to enable the correct termination of transactions.
- Database variables that are not of type relation and that do not contain relation-valued components are mapped to long records (see section 5.7) in a specific database relation. Entries to the data dictionary (layer *CRDS*) are maintained for the handling of these database variables.

- Iterations over relations restricted by a predicate are implemented within this layer. At the commencement of an iteration, all relation elements satisfying the selective access expression are stored in a temporary relation which is then taken as the basis for the iteration itself. This expensive copy process is avoided if the compiler is able to guarantee that the body of the iteration statement is free of updates to the range relation.

- Nested *GetDBVar* and *GetTuple* operations issued in different static contexts by the compiler for the same database object have to be identified dynamically in order to share the same application tuple buffer and to preserve the semantics of traditional variable updates.

- The operations *CreateObject*, *DropObject* and *AssignObject* are intended to simplify the code generation. They help to manipulate composed variables containing relations, selectors and constructors as substructures (e.g. variables of type *Table*, defined on page 32).

5.2 PSMS – Evaluation of Parameterized and Recursive Queries

Since the language DBPL strongly encourages the use of named, parameterized query expressions (selectors and constructors), an important requirement for the DBPL system implementation was to support query optimization also for expressions that involve multiple, independently developed and dynamically bound query expressions. Therefore, the exported *PSMS* operations resemble those found at a standard set-oriented database interface (evaluation of a set-valued or a boolean-valued expression, bulk insertion, deletion, update and assignment). However, since these expressions may contain references to other expresssions and actual parameters to be substituted for the formal parameters of the referenced expression, the expressive power of *PSMS* operations is well beyond relationally complete queries [ERMS91].

On the other hand, selectors and constructors as realized by *PSMS* also support more traditional database system tasks like the

- definition of views and the resolution of queries on views to queries on the underlying base relations;

- definition and check of predicative integrity constraints and access restrictions;

- evaluation of recursive fixed-point queries (as found in deductive databases).

The *PSMS* interface is centered around the ADT *Expression* and provides functions to create elementary expressions from constants and (program or logical) variables, to combine expressions to new expressions (comparison, conjunction, disjunction, quantification) and to introduce references as well as parameters into an expression. *PSMS* operations are provided for symbolic manipulations of expressions (*CopyExpression, DropExpression, StoreExpression, GetExpression, SubstituteWithInPredicate, PrepareForEvaluation*) prior to their evaluation yielding a set-valued (*Evaluate*) or boolean-valued (*BooleanValue*) result.

Values of the ADT *Expression* are implemented as attributed abstract syntax trees that contain pointers to other attributed abstract syntax trees, to global program variables and to

type descriptions. *PSMS Expressions* are evaluated by a mapping to *Predicates* of the module *CPMS* which are in turn evaluated either by *CPMS* routines (*Evaluate, BooleanValue, Assign, Insert, Delete, Update*) or by their counterparts of the layer *SQLGate* that transform these operations into sematically equivalent SQL statements. For non-recursive queries, this mapping can be understood as a simple expansion process that replaces a reference to another expression by a copy of that expression in which formal parameters are subsituted by their corresponding actual parameters. For recursive references between query expressions like in the definition of transitive relationships (e.g., ancestors, transitive subparts, strongly connected components), such a naive expansion process would not terminate. As described in detail in [ERMS91], *PSMS* constructs for a given query Q a graph G that represents the *used by* relationship between named query expressions (more precisely: between parameterized instances of named query expressions) in Q.

Cycles in G correspond to recursive query expressions in Q that have *fixed-point* semantics. Furthermore, each edge in G can be either be marked as "positive" or "negative". Negative edges result from negated or universally quantified subexpressions. It can be shown that a *stratified* [Naq89] recursive query in DBPL corrsponds to a graph G that does not have cycles involving negative edges. If the analyis of a graph indicates a non-stratified query, the transaction that issued the query is terminated with an error message. Otherwise, the graph is partitioned into its strongly connected components G_i that are then evaluated bottom up component by component, replacing evaluated subexpressions (subgraphs) by their relational result. The evaluation of each (cyclic) strongly connected component requires an *iterative* fixed-point computation.

The DBPL system provides two alternative strategies for this fixed-point computation: The naive strategy computes the fixed-point of a set of recursive set expressions starting with the empty set and by repeated application of the set expressions to the result derived in the previous iteration. The preferred *PSMS* strategy is to apply a delta transformation [GKB87] to the set expressions prior to their repeated evaluation. Although this symbolic transformation increases the complexity of the set expressions, it typically reduces the evaluation time by an order of magnitude. Essentially, this "wave-front" optimization simply avoids the redundant recalculation of a large number of result tuples in consecutive iterations by exploiting the monotonicity of stratified queries. The naive evaluation strategy is only employed in "pathological" cases where the delta transformation would result in an exponential blow-up of the number of relations involved in multi-way joins.

As mentioned above, *PSMS* makes heavy use of query optimization and query evalaution functions exported by the layer *SQLGate* . Since *PSMS* typically re-evaluates a given non-recursive query expression several times in short succession, it turned out to be advantageous to have separate functions for the symbolic optimization of a query expression against a given database (*CPMS.Transform*) and its evaluation (*CPMS.Evaluate*).

5.3 CTMS and LMS – Multi-Level Transaction Management

In a multi-user environment, operations on persistent objects have to be synchronized against each other. In DBPL the unit of concurrency control and recovery is the transaction.

The DBPL system utilizes a *three-level* synchronization scheme (indicated by the arrows to module *LMS* in Fig. 1). Serializability and recovery of (flat) user-defined DBPL transactions

is achieved by appropriate *CTMS* locking and logging strategies at the abstraction level of complex objects and set-oriented expressions over these objects. The design decision to put *CTMS* below *PSMS* considerably simplifies the structure of expressions to be analyzed by the scheduler (i.e. no recursion, no references to other expression) while maintaining a sufficient high level of abstraction (essentially relational calculus expressions) to support advanced concurrency control mechanisms (like predicative locking or validation).

CTMS synchronization and recovery works under the assumption that individual *CRDS* operations (like *InsertTuple, GetTuple*) are executed *atomically* and that conflicts arising from the concurrent use of access paths or from specific page allocation strategies for complex objects are also handled internally by the layer *CRDS*. *CRDS* operations are therefore *nested transactions* and require appropriate locking and logging mechanisms [BSW88, Wei88]. In fact, this division of labour between higher and lower-level transactions can be also found between the layers *CRDS* and *SMS* since *SMS* operations are again executed atomically and recoverable.

The foremost advantage of such a nested transaction scheme is its strong support for flexible, modular database system architectures since higher-level transactions can abstract from the implementation details of lower-level transactions and transactions on each layer can exploit local knowledge about possible concurrently executing transactions on their abstraction level. For example, the layer *CRDS* is capable of avoiding deadlocks by acquiring locks on *CRDS* objects in a commonly agreed order.

Since locks of lower-level (*CRDS, SMS*) transactions are already released at the end of a subtransactions, another advantage of nested transactions (often quoted in the literature [BSW88]) is a gain in parallelism for massive multi-user applications. In DBPL, however, the increased parallelism does not yield a corresponding increase in total transaction throughput since nested transactions introduce some bookkeeping overhead (e.g. there are three logs and lock requests on three distinct layers for a given user-level operation).

The layer *LMS* provides the generic services required for the implementation of transactions (handling of a *write-ahead-log*, distribution of lock requests from application programs to the centralized scheduler, generation of lock identifiers). Each of the layers *CTMS, CRDS* and *SMS* specializes these services for its own purposes: *CTMS* maintains a wait-for graph to detect deadlock situations and utilizes a multi-granularity locking scheme [GLP75], while the index management in the layer *CRDS* employs a tailored graph locking protocol for B-link trees [LY81] that supports concurrent updates. Page and record operations are synchronized using a standard strict two-phase locking scheme.

The current DBPL system does not provide crash recovery since all log records are not forced to stable storage but are simply kept in main memory.

5.4 SQLGate – Set-Oriented Access to Internal and External Databases

The layer *SQLGate* abstracts from the details of the optimization and evaluation of set-oriented queries against DBPL and SQL databases. Higher levels of the DBPL system and DBPL programmers can transparently access and manipulate internal as well as external database objects. In particular, it is possible to write expressions that freely combine objects from both worlds. At runtime, all references from DBPL programs to external databases are

transformed into dynamic SQL+C statements according to the X-Open standard. All capabilities of SQL servers like query optimization, concurrency control, access path management, access control and distribution are therefore automatically utilized by DBPL programs.

Since the layer *SQLGate* is below the transaction, selector and constructor management layers of DBPL, exception handling, integrity checking and fixed-point queries are already mapped to "simple" concepts like flat transactions or set-oriented queries and updates. All remaining constructs of DBPL addressing SQL relations are converted into sequences of calls of procedures from a single module written in C and embedded SQL. For example, the DBPL expression ("Suppliers supplying all parts")

```
EACH X IN supp: ALL  Y IN part ( SOME Z IN sp (
                (X.sno = Z.sno) AND (Y.pno = Z.pno) ) )
```

is mapped into the following SQL query:

```
select * from supp X1
where not exists (
            select * from part X2
            where not exists (
                        select * from sp X3
                        where X1.sno = X3.sno
                        and   X2.pno = X3.pno ) )
```

This query is executed on the SQL server and, depening on its context, the result is further processed by DBPL or SQL.

The generation of a SQL statement for a given DBPL predicate is done by a recursive scan of the DBPL predicate tree. During the scan, a list of SQL lexicals is created. Roots of the tree corresponding to access expressions cause the insertion of the lexicals *select*, *from* and *where* into the list. Next, projections in the tree insert proper lexicals after *select*, range relations in the tree insert proper lexicals after *from*, and conditional expressions insert proper lexicals after *where*. The list is organized as a stack: to handle nested select blocks correctly, insertions are done behind the *select*, *from* or *where* literal that is found at the list head. When the select block is completed, it is "masked" and thereby invisible for further insertions. A similar algorithm is employed for *exists* and other SQL lexicals to cover all situations that can occur in DBPL predicate trees. The final SQL query is obtained as a simple concatenation of the lexical list after the recursive traversal.

Predicates that mix DBPL and INGRES relations are allowed. The corresponding translation procedures recursively scan a predicate tree and produce one of the following answers: *pure dbpl, pure ingres, top dbpl, top ingres, mixed joins, badly mixed.* The answer *pure ingres* means that the predicate contains references to SQL relations only, and can be converted into an SQL query. *top dbpl* means that the predicate contains independent subpredicates that are *pure ingres.* They can be evaluated completely on the server and then the whole predicate becomes *pure dbpl.* Similarly, *top ingres* means that the predicate contains subpredicates that are *pure dbpl.* They can be evaluated completely on the side of DBPL (using the services of *CPMS* and the resulting temporary relations are then copied to the SQL server. Thus, the whole predicate becomes *pure ingres.* In the case of *mixed joins*, the participating DBPL

relations are sent to the SQL server and the predicate becomes *pure ingres*. The result *badly mixed* means that all good methods fail and the only method is copying SQL relations to the DBPL side. For performance reasons, in the current implementation this situation raises a run-time error.

Based on similar principles, all other constructs of DBPL addressing external databases, in particular the high-level relational assignments and updates, *for each* statements, relational comparisons, and low-level features are automatically converted into calls of C+SQL procedures which utilize standard SQL DBMS functions.

5.5 CPMS – Transformation and Evaluation of Complex Object Queries

Essentially, *CPMS* is composed of two components: *Evaluation System* and *Transformation System*. Both components deal with problems created by allowing predicates over type-complete data objects. The major interdependences between the two components are based on the fact that the predicate transformation module is aware of the needs of the evaluation module and transforms predicates into a structure that is considered advantageous for its evaluation.

The layer *CPMS* exports (1) data structures for the internal representation of predicates, (2) equivalence transformations on predicates to achieve a standardized predicate structure or an improved query evaluation efficiency, and (3) evaluation routines for quantified boolean predicates [**some/all** *r* **in** *rel (predicate)*], set-valued expressions [**each** *r* **in** *rel: predicate*], and the test of the validity of a predicate for a given relation element *tup* [**some/all** *r* **in** *rel (predicate(r, tup)*].

A predicates is represented by a predicate tree. Nodes are used to define the elements of a predicate, whereas edges describe its syntactical structure. Fig. 11 sketches the predicate tree for the following single-variable query:

```
{{x.Delivery.Day, x.Delivery.Month} OF EACH x IN A[20]: x.Delivery.Year=1988
```

Eight different node types suffice to represent arbitrary complex DBPL queries: *constant* nodes appear in terms and projection lists; *index* nodes represent array accesses; *variable* nodes can be either bound to quantifiers (some, all, each) or to a global program variable (variables are identified internally by numbers that are unique within a given expression); *projection* nodes are used in the target list of a query or in the definition of a key access; *term* nodes are further distinguished into monadic, dyadic and boolean terms; *connection* nodes represent disjunctions or conjunctions; *quantification* nodes introduces scopes for bound relational variables; *root* nodes correspond to relational expressions. A root node contains the hierarchical type description of the relational result type. A root node can represent an empty relation, a relation variable, a selective access expression, a selector or constructor, or a single relation element.

PSMS provides several algorithms for the transformation of predicate trees into semantically equivalent trees. Some of them (elimination of negations, elimination of empty ranges, constant folding, transformation into prenex normal form) aim at a standardization, simplification and decomposition of queries in order to simplify the subsequent query evaluation or query optimization process. Other transformations are used for query optimization tasks.

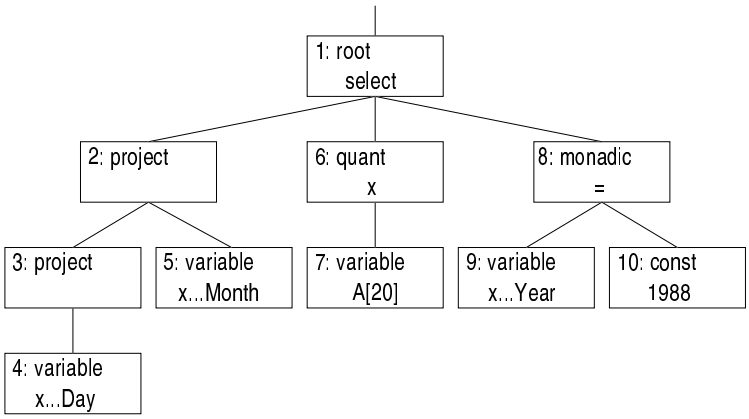


Figure 11: Predicate tree for a DBPL query expression

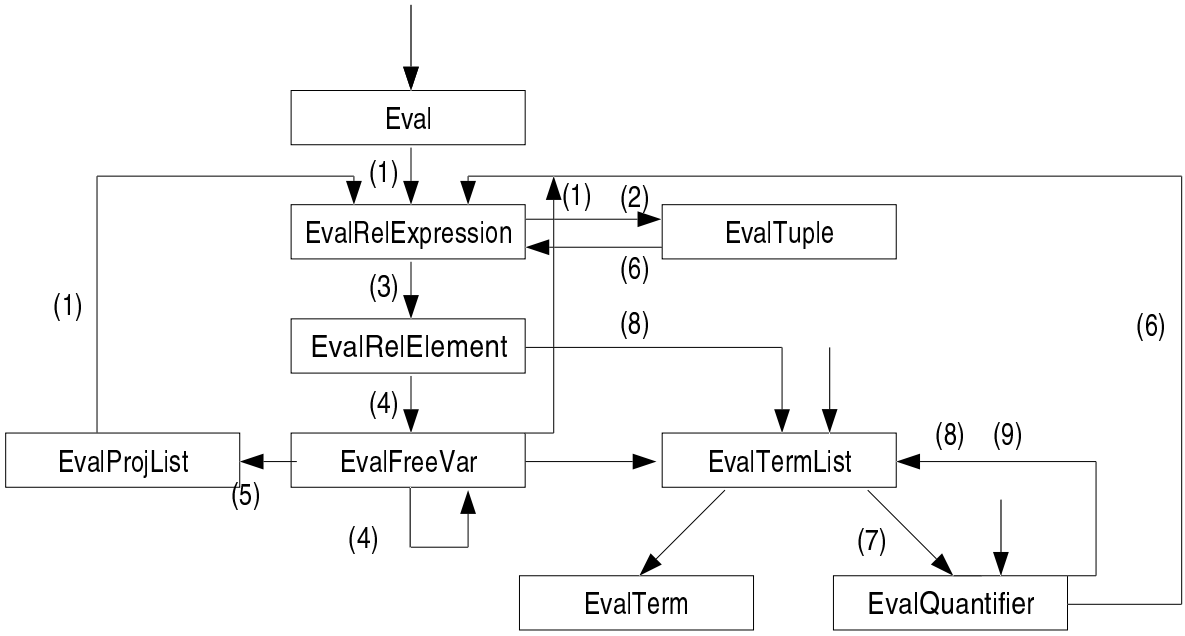
Some of them employ algebraic equivalences (propagation of filters and projections over joins and unions), others introduce implementation-oriented access mechanisms (primary key, secondary key or hierarchical access) into the query representation.

The *PSMS* transformation system also utilizes cardinality information about relations involved in a query expression. Thereby, one can distinguish between data-independent and data-dependent transformations. Even though the former are executable by the compiler, the DBPL system carries out all transformations at run time. Although this leads to an additional expense during runtime, there are also some advantages: the compiler can work with a simplified internal structure of predicates; interactive components can pass user-defined predicates on to the runtime system without preceding transformations; all transformation routines and data structures for predicate trees are localized in a single component of the database system.

The main strategies of the current DBPL query optimizer to efficiently deal with complex objects are to minimize the read set of a query, to exploit the primary key access structures maintained in the layer *CRDS* not only for flat relations but also for subrelations in complex objects, and finally to minimize the repeated re-evaluation of target expressions involving relational subqueries. Furthermore, *CPMS* can rely on powerful complex object operations provided by *CRDS* to efficiently access and copy complete substructures of hierarchically structured objects. For flat relations, many of the algorithms developed for the Pascal/R system and previous DBPL system versions are still employed [JK83, Koc84, JK84].

Fig. 12 may give an idea of the highly recursive structure of the evaluation procedures for complex object queries that is implied by the orthogonality of type constructors and query expressions in DBPL. The evaluation procedures are not only capable of evaluating set-oriented expressions, but they can be also employed to evaluate boolean-valued quantified expressions and to check integrity constraints on individual relation elements (as required by the layer *PSMS*).

As a first cut, the central evaluation algorithm of the *CPMS* evaluation system can be un-



(1) relational expression

(2) tuple

(3) selective / constructive expression

(4) quantified variable

(5) projection list

(6) nested range expression

(7) quantified range

(8) independent term

(9) dependent term

(10) boolean, monadic or dyadic term

Figure 12: Evaluation procedures for DBPL queries and their mutual dependencies

derstood as a *nested loop* algorithm extended to handle relational subqueries and target expressions. The algorithm employs sophisticated bookkeeping mechanisms (so-called "virtual" conjunctive and disjunctive normal forms and arrays of bit-sets of modified loop variables) to re-use partial results computed in earlier iteration steps. Furthermore, sequential scans over (sub)relations can be replaced by value-oriented (primary or secondary) key accesses provided by *CRDS* operations (*FindKey, FindRange*).

5.6 CRDS – Type-Complete Relational Database Management

Using the facilities for fixed-sized short records and variable-sized long records exported by the storage management system *SMS*, *CRDS* offers an external, abstract view on complex objects of the DBPL type-complete data model. *CRDS* implements the ADTs *Type*, *Relation*, *Key* and *Database* used by the upper DBPL system layers.

The primary concern for the design of the *CRDS* interface was to achieve the complete functionality for all kinds of relations in a uniform way. In particular, this includes the possibility of selective and associative access to (nested) relations of arbitrary depth. The interface offers the following services:

- Creation of type structures;
- Creation, opening and closing of root relations;
- Monadic and dyadic relation operators applicable to arbitrary relations and combinations thereof (*ClearRel*, *Card*, *Empty*, *AssignRel*);
- Navigational and direct access via the primary key (*FindFirst*, *FindNext*, *Find*);
- Retrieval of relation elements or parts thereof (*GetTuple*);
- Modification of relation elements (*InsertTuple*, *UpdateTuple*, *DeleteTuple*);
- Definition of secondary access paths for root relations and their employment in conventional and non-standard search routines (*FindFirstKey*, *FindKey*, *FindRange*);
- Procedures for the handling of variable-sized long attributes (*GetLongField, InsertInLongField, DeleteFromLongField, UpdateLongField*);
- *BOT*, *EOT* and *UNDO* operations.

The module *CRDSDatabases* offers additional features for manipulating databases (create, open, drop) and *data dictionaries*. Data dictionaries are implemented as relations of tuples with variable-sized long attributes and are also made available to higher levels of the DBPL system for their private purposes.

Relational data structures are implemented in the layer *CRDS* as follows: tuples of first normal form ("flat") relations are mapped directly onto fixed-sized *SMS* records. If the tuple size exceeds the maximum page size of the underlying operating system (as defined at DBPL installation time, e.g. 4K), *CRDS* automatically maps these tuples onto page-spanning long records. In both cases, tuples are identified via stable tuple identifiers (*TID*s) and a B-link

(Map-Id)	Rel-Id	Parent-Id	Brother left	Brother right	Key-Part	Data-TID
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 13: Structure of a map

tree is utilized to enforce the primary key constraint and to speed up direct and sequential access to relation elements. The B-link tree is an extension of the B*-tree, efficiently solving the problems given by concurrent operations on this kind of data structure [LY81]. The DBA can define dynamically additional secondary indices for relation variables that are also maintained by *CRDS* operations.

CRDS utilizes a *key-oriented chained map concept* for a compact representation of the structure of NF^2relations. This storare concept is a substantial extension of the map structures presented in [LKM+84] to provide fast indexed, sequential and key-based access to (nested) relations of arbitrary depth. It is based on a separation between user information and structural information:

- Complex relation elements are decomposed by a concatenation of all non-relational, fixed-sized attributes at the different levels, storing them together as a flat *SMS* (long or short) record. The dissection starts at the top level, and nested relations are then decomposed recursively.
- A data structure called *map* is associated with each root element, containing information about the relationship and the key-based order of the (nested) relation elements. They can be accessed by their storage identifier (*TID*), also contained within the map.

A map is implemented as a *SMS* long record and is interpreted a vector of numbered entries (Map-Id, see Fig. 13). Each entry corresponds to exactly one of the (nested) relation elements obtained during the decomposition. The different columns of the map have the following meaning:

Rel-Id: Nested relations are uniquely numbered within each NF^2relation type;

Parent-Id: Reference to the map entry of the parent tuple that contains this element;

Brother left/right: Reference to the entry of the brother tuple with the next lower or higher key value;

Key-Part: A fixed-sized key value prefix of the corresponding tuple;

Data-TID: The storage identifier of the atomar fragment of the tuple.

Whereas a pair [Rel-Id, Parent-Id] uniquely identifies a nested relation, thus being useful to model the hierarchical relationship, the two brother columns — constituting a doubly connected list structure — function as a (sequential) access path to each of the nested relation elements. Their performance is enhanced by the key prefix which in most cases prevents data from having to be accessed. As, apart from the pair [MapTID, MapIndex], elements of nested relations are addressed indirectly, this *SuperTID* represents a stable database address.

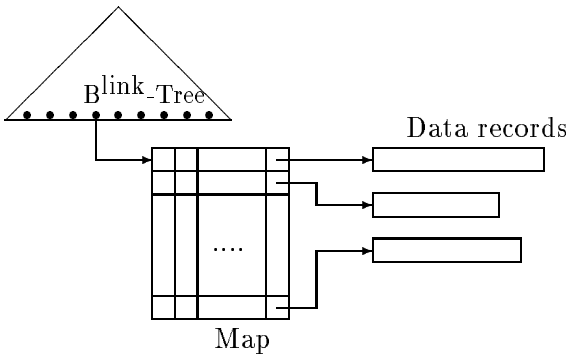


Figure 14: *CRDS* access path management

As illustrated in Fig. 14, a NF^2relation is implemented by a B-link tree that functions as an index to maps which in turn provide access to all the data records that make up an element of the root relation.

5.7 SMS – Persistent Storage Management

The layer *SMS* performs rather traditional DBMS system tasks like record allocation, record identification, buffer management and free space management. In addition to fixed-sized data records, *SMS* also exports page spanning long records. Long records are of dynamical and almost unrestricted size and allow partial retrieval and modification. A long record is organized by means of a directory storing the length and the address (TID) of all short records belonging to it. The directory itself is also implemented as a short record. This implies that the length of an individual long record is bounded by $|\text{page}|^2/|\text{TID}| + |\text{Length}|$. Typical page sizes between a half and 4K lead to a maximum length of 43K – 2.8MByte.

An important taks of *SMS* in the DBPL system is to provide *persistence abstraction*: *SMS* clients need not to be aware whether record operations are executed locally in main memory data structures or on a remote machine on persistent data structures. The only difference lies in the fact that only record operations on shared data structures have to be executed atomically and recoverable.

Since DBPL supports client-server architectures and client machines have their own page buffers, there is a need for a cache coherence protocol between concurrently executing clients. An important optimization to minimize network traffic and to significantly speed up remote database accesses is achieved by "piggy packing" the time stamps required by the cache coherence protocol to higher-level lock messages sent to the central *LMS* lock server (see Fig. 1).

In contrast to other database management systems, the DBPL system makes heavy use of the possibility to distribute disjoint database objects to different operating system files. This complicates the internal identification of data records and the free space management, but simplifies database evolution, backup and access control using operating system programs.

6 Using the DBPL System

The DBPL project always had a strong commitment to implementability. A multi-user DBPL system under VAX/VMS has served many times since 1985 for lab courses on database programming at the Universities of Frankfurt and Hamburg. There exist several DBPL system extensions that experiment with alternatives for concurrency (optimistic, pessimistic and mixed strategies) [BJS86] and integrity control [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, for example, a re-implementation of the DBPL compiler to generate native code for IBM-PC/AT clients in cooperation with VAX/VMS servers.

In 1991, a substantial effort was made to integrate the experience gained with these prototypes into a new, portable implementation of the DBPL runtime system on various platforms (VAX/VMS, Sun-3, Sun-4/Unix, IBM RISC/AIX). By utilizing Sun's optimizing compiler backend, the DBPL compiler achieves "production-quality" performance and interoperability.

The availability of an optimizing and transparent gateway from DBPL to SQL database servers since early 1992 substantially increases the attractiveness of DBPL for users that have to work with large, possibly pre-existing databases and that require well-established tools for access control, data clustering, interactive database access etc. and last, but not least, interoperability with non-DBPL database applications.

To summarize, we expect the DBPL language and system to be used in research and devleopment mainly for the following three tasks:

Concept validation: As outlined above, we strongly believe in the necessity of experimental evaluation of proposed system solutions (e.g. of new concurrency control protocols or a new workstation-server architectures). In many cases, the interaction with several system components (e.g. the recovery management or the query optimizer), or the lack of universality severly impairs the utility of a seemingly advantageous paper-and-pencil solution.

Database education: Our experience in using DBPL intensively in lab classes convinces us that it is an appropriate tool for teaching the essential problems and solutions in database application development. Without being distracted by idiosyncratic surface syntax and deficiencies of traditional preprocessor database interfaces, it is much easier to isolate and communicate the repeating patterns in database applications and to concentrate on an *abstract and complete* picture of database application programming.

Application prototyping: From Modula-2 the DBPL language has inherited software engineering qualities that can not be found in commercial database environments and which qualify DBPL as an appropriate tool for designs and implementations. This use of DBPL is further supported by the quality of the commercial platforms on which the DBPL system is realized, the depth of its integration into professional environmets for software development and maintenance and its interoperability with commercial relational database servers.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[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.

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

[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.

[C+86] M. Carey et al. The Architecture of the EXODUS Extensible DBMS. In *Proc. International Workshop on Object-Oriented Database Systems*, pages 52–65, Pacific Grove, Ca., September 1986.

[ERMS91] J. Eder, A. Rudloff, F. Matthes, and J.W. Schmidt. Data Construction with Recursive Set Expressions in DBPL. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

[GKB87] Ulrich Güntzer, Werner Kiessling, and Rudolf Bayer. On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration. In *Proceedings 3rd International Conference on Data Engineering*, pages 120 – 129, Los Angeles, February 1987.

[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.

[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 *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 196–206, May 1983.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[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.

[KMP82] J. Koch, J. Mall, and P. Putfarken. Modula-2 for the VAX: Description of a System Portation. In H. Langmaack, B. Schlender, and J.W. Schmidt, editors, *Tagungsband Implementierung Pascal-artiger Programmiersprachen*. Teubner Verlag, 1982. (in German).

[Koc84] J. Koch. *Relationale Anfragen: Zerlegung und Optimierung.* PhD thesis, Fachbereich Informatik, Universität Hamburg, West Germany, December 1984.

[LKM+84] R. Lorie, W. Kim, D. McNabb, W. Plouffe, and A. Meier. Supporting Complex Objects in a Relational System for Engineering Databases. In W. Kim, D.S. Reimer, and D.S. Batory, editors, *Query Processing in Database Systems*, pages 145–155, Berlin, 1984. Springer-Verlag.

[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.

[MS92] F. Matthes and J.W. Schmidt. The Database Programming Language DBPL: Rationale and Report. FIDE Technical Report FIDE/92/46, Fachbereich Informatik, Universität Hamburg, West Germany, July 1992.

[Muc90] S.S. Muchnick. Optimizing Compilers for the SPARC Architecture. In M. Hall and J. Barry, editors, *The Sun Technology Papers*. Springer-Verlag, 1990.

[Naq89] S.A. Naqvi. Stratification as a Design Principle in Logical Query Languages. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, June 1989.

[PSS+87] H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum, and U. Deppisch. Architecture and Implementation of the Darmstadt Database Kernel System. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pages 196–207, May 1987.

[RA83] M. Reimer and Diener A. The Modula/R Compiler for the Lilith. LIDAS Memo 051-83, Department Informatik, ETH Zürich, Switzerland, 1983.

[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.

[SM90] J.W. Schmidt and F. Matthes. Language Technology for Post-Relational Data Systems. In A. Blaser, editor, *Database Systems of the 90s*, volume 466 of *Lecture Notes in Computer Science*, pages 81–114, November 1990.

[SM91a] J.W. Schmidt and F. Matthes. Modular and Rule-Based Database Programming in DBPL. FIDE Technical Report FIDE/91/15, Fachbereich Informatik, Universität Hamburg, West Germany, February 1991.

[SM91b] J.W. Schmidt and F. Matthes. Naming Schemes and Name Space Management in the DBPL Persistent Storage System. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts*. Morgan Kaufmann Publishers, January 1991.

[Sto90] M. Stonebraker. Special Issue on Database Prototype Systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

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

[Wei88] G. Weikum. *Transaktionen in Datenbanken: Fehlertolerante Steuerung paralleler Abläufe*. Addison Wesley, 1988.