Appeared in: *Information Systems*, Vol. 19,No. 2, pp. 121–140,1994

The DBPL Project[1]: Advances in Modular Database Programming

Joachim W. Schmidt Florian Matthes

Universität Hamburg
Department of Computer Science
Vogt-Kölln Straße 30
D-22527 Hamburg, Germany
{J_Schmidt,matthes}@dbis1.informatik.uni-hamburg.de

Abstract

In the DBPL project we tackled the problem of supporting data-intensive applications in a single framework, clean and simple in its conceptual foundation and free of technical mismatches. Conceptually, we based the DBPL language on Modula-2 with three built-in extensions which, at that time, were considered necessary (and sufficient) for data-intensive applications:

- a parametric bulk type constructor for "keyed sets" or relations;
- a module concept which supports sharing across programs and implies persistence;
- a procedure concept with transactional semantics, i.e., serializability and recovery.

In implementing the DBPL system we adopted a rather strict approach by aiming for

- full orthogonality in our relationally extended type space;
- type-complete persistence, i.e. longevity of data from Booleans to relations;
- functional abstraction for relational expressions including recursion.

As a consequence, the DBPL project covers in a quite natural way approaches otherwise considered disjoint as, for example, complex objects, multibases and data deduction.

In this paper, besides reporting on project achievements and spin-offs, we also present our insight in good-quality project design and its dependency on conceptual simplicity and implementational strictness. This also includes early recognition of the essential project "terminator" which, in our project, was Modula-2's monomorphic type system and its intrinsic restrictions on systems extensibility.

[1]This research has been supported in part by ESPRIT Basic Research, Project #892 (DAIDA), #3070 (FIDE), #6309 (FIDE$_2$).

The DBPL project results in a mature product[2] for modular database application programming and a follow-up project based on polymorphic types and other higher-order concepts[3].

1 Introduction

Our research in database programming languages started in the mid 70ies when we first suggested some high-level language constructs for data of type relation [1]. The prime goal of the related R&D projects was to understand better and relate the data modeling requirements of programs and databases and to support data-intensive application development by systems free of conceptual and technical mismatches.

This research can be divided clearly into two series of projects, reflecting both the increasing demands on database programming but also our better understanding of the field. The first run of projects, which is evaluated in this paper, started with Pascal/R [1, 2, 3] and culminated recently in the delivery of a mature implementation of the DBPL system [4, 5, 6, 7, 8]. Here database programming support is based on *monomorphic type systems* resulting in a fixed selection of type constructors which is hard-wired into DBPL and laid out to span a wide type space which also includes bulk data types. Around 1990 we started transferring our DBPL experience into a new set of projects based on *polymorphic type systems* which enable add-on bulk type definitions [9] and a more open approach to system construction. This ongoing experience is referred to briefly at the end of the paper.

For DBPL we chose Modula-2 [10], a modular system programming language, as its algorithmic core. Our development of specific language extensions required by data-intensive applications (such as bulk types, iteration abstractions, transactions and persistence abstractions) was guided by a strict adherence to the widely accepted language design principles of *simplicity*, *orthogonality* and *abstraction* [11]. This is to be seen in direct contrast to some DB-language developments – notably SQL – which appear to be driven by ad-hoc definitions and series of local extensions.

The paper is structured as follows: in section 2 we review the rationale behind our design of DBPL that sets DBPL apart from extended database query languages, persistent programming languages and conceptual modeling languages. We then present the essential DBPL language concepts that support bulk data structuring, persistence abstraction and declarative bulk data manipulation (Sec. 3). In section 4 we shed some light on the extensions to relational database and procedural programming language technology that we had to develop to support DBPL's orthogonal and fully integrated language model.

Due to the longevity of the DBPL project (the first language report appeared in 1985 [12], the most recent version of the system was released at

[2]The DBPL System is distributed through Hamburg University.

[3]Tycoon: Typed Communicating Objects in Open Environments.

the end of 1992 [13]), it is interesting to review briefly the evolution of DBPL (Sec. 5.1) and to highlight spin-off research projects (Sec. 5.2) that relied heavily on the availability of DBPL as a stable, algorithmically complete database programming language. These spin-offs expanded into the area of distributed systems, interoperability and database design environments. We conclude in section 6 by justifying our DBPL project decision to "declare success and quit" – keeping in mind DBPL's goals and means.

2 The Rationale behind DBPL

Data-intensive applications are characterized by their needs to model and manipulate *heavily constrained* data that are *long-lived* and *shared* by a user community. These requirements result directly from the fact that databases serve as (partial) representations of some organizational unit or physical structure that exist in their own constraining context and on their own time-scale independent of any computer system. Due to the size of the target system and the level of detail by which it is represented, such representational data may become extremely voluminous. In strong contrast to the need for global management of large amounts of *representational data*, data-intensive applications are also required to process small amounts of local *computational data* that implement individual states or state transitions.

In essence, it is that broad spectrum of demands – the difference in purpose, size, lifetime, availability etc. of data – combined with the need to cope with all these demands within a single conceptual framework that is decisive in the design of a database programming language [14].

In DBPL [6, 7, 8], we address the above issues by incorporating a set- and rule-based view of relational database modeling into the well-understood system programming language Modula-2.

2.1 Conceptual DBPL Foundations

The leitmotiv behind the DBPL design was the exploitation and *integration* of a solid, well-understood conceptual framework capable of capturing consistently the wide range of data-intensive application requirements — and not the desire to implement new theoretical concepts in isolation.

Classical *type systems* and the *relational data model* are the cornerstones of the DBPL language design. Both contributions provide an enlightening foundation from a technological as well as from a theoretical point of view. However, the revealing new experiences are made at the border where types (and expressions, selectors, functions ...) and data models (and queries, views, transactions ...) meet and have to be understood, one in light of the other.

The DBPL language and system both rely heavily on conceptual and theoretical input from the following research areas: programming language foundations (typing, scoping, binding; [8, 15]), compilation technology (type checking, local data flow analysis, intermediate languages, separate compilation; [16, 17]), extended relational models (data constructors, bulk operations,

query languages and their expressive power), query optimization (transformations at compile- and run-time, cost models, search strategies; [18]), recursive query evaluation (fixed point semantics, stratification, recursive rule definition; [19]), concurrency control (multi-level concurrency control and recovery for complex objects; [16]), and distribution models (transaction procedures, sagas, compensating transactions; [20, 21]).

As indicated in the preceding references, our DBPL research contributed to the further development of some of these conceptual and technological foundations. In general, however, one can say that the DBPL project stayed more or less inside the boundaries drawn by conventional language technology and concentrated on improving the linguistic support for state-of-the-art database technology.

2.2 Language Design Considerations

Our design of DBPL can be characterized by the phrase "power through orthogonality". Instead of designing a completely new language with its own naming, binding and typing rules, we extended an existing language (Modula-2 [10]) and placed particular emphasis on the interoperability of the new database concepts with the given programming language concepts. At DBPL design-time we considered the following extensions necessary (and sufficient) to meet the specific requirements of data-intensive applications:

- bulk data structuring using an orthogonal type constructor `RELATION OF` (see Sec. 3.2);
- abstraction from persistence, sharing, concurrency control and recovery based on the concept of `DATABASE MODULE`s and parameterized `TRANSACTION`s (see Sec. 3.3);
- abstraction from bulk iteration through associative *access expressions* (see Sec. 3.4);

In particular, we eliminated the traditional competence and impedance mismatch [22] between programming languages and database management systems by providing for

- a uniform treatment of volatile and persistent data ("orthogonal persistence" [23]),
- a uniform treatment of large quantities of objects with a simple structure and small quantities of objects with a complex structure ("type completeness" [23]), as well as
- a uniform (static) compatibility check between the declaration and the utilization of each name ("strong static typing" [24]).

Implementation details such as the storage layout of records, the clustering of data, the existence of secondary index structures, query evaluation strategies, concurrency and recovery mechanisms, are deliberately hidden from the

DBPL programmer. A key idea in the design and implementation of DBPL is to let the system choose appropriate implementation strategies based on high-level information extracted from the application program or its environment. As it turns out, the widespread use of access expressions (i.e. first-order logic abstractions of bulk data access) in typical DBPL programs facilitates such an approach.

DBPL reflects Modula-2 by occasionally sacrificing "language orthogonality" based on engineering and efficiency considerations. For example, DBPL does not support persistence of pointers and procedure variables, concepts, which in some sense provide the technological basis for object identification and procedural attachment. This was not only motivated by the predominance of associative identification mechanisms in the classical Relational Data Model, but also by the far-reaching consequences of this identification mechanism on the concurrency control and distribution support [25, 21].

Modula-2 was chosen as the basis for DBPL because of its software engineering qualities. It provides a clear module concept and a strict type system and exceeds further through its balance between simplicity and expressive power. Finally, our group has a long tradition in Modula-2 compiler construction [26, 27].

The compatibility between DBPL and Modula-2 is a design decision with far-reaching consequences. DBPL is designed to be fully upward compatible with Modula-2, i.e. every correct Modula-2 program has to be a correct DBPL program. This decision not only limits the freedom in language extensions (for example, the keyword *SET* is already used for bit sets in Modula-2 and is not available for "true" sets, i.e., relations in DBPL), but also forces adherence to language mechanisms (variant records, string handling, local modules, ...) for which nowadays "better" solutions are available. The main advantage of our compatibility decision is to lower the conceptual and technological burden for DBPL users since they do not have to learn yet another language. Furthermore, this decision allows the smooth integration of DBPL into existing, fully-fledged software development environments (debuggers, profilers, version managers, ...), and the benefit of software libraries and of interfaces to other languages and systems.

In contrast to some persistent programming languages, the evolution of database schemata and of application programs is left outside the scope of DBPL. This was based on the insight that this is a complex issue in itself that should be delegated to a specific design environment (see Sec. 5.2.3).

3 DBPL Language Concepts

The first step towards a better integration of database concepts into a language environment is to identify the basic database concepts (like domains, tuples, tables and schemata) and to rephrase them in terms of an appropriate vocabulary of programming concepts (like types, type constructors and modules). In the following, we illustrate how DBPL captures the main principles

of set- and predicate-oriented database modeling using the well-understood notions of naming, typing and binding in procedural programming languages.

3.1 Uniform Naming and Typing

Names in DBPL are arbitrarily long sequences of upper- and lowercase letters and digits starting with a letter. Names are case sensitive and are used to identify semantic objects of the language (values, types, variables, procedures, transactions, modules, ...).

DBPL is a *statically and strongly typed monomorphic* language: every name is associated with a unique type that is determined at compile-time. The compiler uses this information to assure that all names for values, expressions or operations are only used in an appropriate context. The advantages of such a typing scheme are well known: programs are less liable to contain errors and there are no time-consuming dynamic type checks.

The type compatibility rules for composite types in DBPL are based on *name equivalence*, i.e., two composite objects have the same type if and only if they have been declared by using the same type name. This should be seen in contrast to the rule of *structural equivalence* where two objects are type compatible if their fully expanded definitions are isomorphic. Most relational systems do not allow the introduction of names for relation types and schemata and are therefore based on structural equivalence.

DBPL provides a rich set of *built-in types* like `INTEGER`, `LONGINT`, `CARDINAL` (natural numbers), `LONGCARD`, `BOOLEAN`, `CHAR`, `REAL`, and `LONGREAL` that gives programmers a fine control over the mapping from application-specific domains to tailored machine representations:

```
TYPE Dollar = REAL;
```

Values of these built-in types are denoted by predefined names (e.g., `TRUE`) or literals (e.g., `0`, `3.2E-2`, `"A"`). It is furthermore possible to introduce user-defined names for constant values:

```
CONST DollarToDMRatio = 2.64;
```

DBPL supports application-specific extensions of the set of basic types. An *enumeration type* is defined by a list of named values:

```
TYPE SupplierStatusType = (unimportant, important, veryImportant);
```

This declaration defines an order on the values of the enumeration type:

```
    unimportant < important < veryImportant
```

Subrange types are derived from either basic or enumeration types. They impose additional restrictions on the range of values described by their base type:

```
TYPE PartNumType = [0..99999];
     ImportantStatusType = [important..veryImportant];
```

Within expressions, values of a subrange type are compatible with values of their base type. Assignments of base type values to variables of a compatible subrange type are checked at run-time.

Strings are treated as composite objects consisting of a sequence of characters. Their type is therefore `ARRAY [0..maxlength] OF CHAR`, where `maxlength` can vary from one string type to another.

```
TYPE LongString = ARRAY [0..999] OF CHAR;
     String = ARRAY [0..29] OF CHAR;
```

Arrays are composed of a fixed number of elements. These elements are positionally designated by indices, which are values of the index type. The latter must be an enumeration type, a subrange type or the basic type `BOOLEAN` or `CHAR`.

```
TYPE BonusTable = ARRAY SupplierStatusType OF Dollar;
```

A *record type* declaration defines a labeled cartesian product type. The scope of the label names is the record definition itself. These names are also accessible as field designators referring to components of variables of that record type (e.g. `supplier.Name`).

```
TYPE SupplierRecType = RECORD
                         Num     : SupplierNumType;
                         Name    : String;
                         Status  : SupplierStatusType;
                       END;
```

3.2 Orthogonal Relation Types

A *relation type* specifies a structure consisting of elements of identical type, called the relation element type. The number of elements, called the cardinality of the relation, is not fixed. The declaration of the relation type specifies the relation element type (e.g., `SupplierRecType`) and an ordered list of key components (e.g., the field `Num` of `SupplierRecType`):

```
TYPE SupplierRelType = RELATION Num OF SupplierRecType;
```

The classical relational data model constrains the relation element type to records with attributes of the built-in types ("flat tuples"). By lifting this rather ad-hoc restriction, we substantially enhanced the modeling power of the language and obtained structuring capabilities beyond data models that simply support *complex objects* and non-first-normal-form relations [28]. For example, the following DBPL types are also well formed:

```
TYPE PartNumSetType     = RELATION OF PartNumType;
     Point2DSetType     = RELATION OF ARRAY [1..2] OF REAL;
     GroupedSuppRelType = RELATION Num OF RECORD
                            Num       : SupplierNumType;
                            SuppParts : PartNumSetType;
                          END
```

This type declaration introduces three type names to denote sets of natural numbers, sets of points that are represented by their coordinates in the plane, and nested relations that group supplier numbers with sets of part numbers.

The *relation key* in a relation type expression defines a list of components within the relation element type such that the relation always defines a (partial) function between its key and its element type. In other words, each key value uniquely determines (at most) one relation element. For example, the key constraint for a relation `SupplierRel` of type `SupplierRelType` can be expressed by the following predicate stating that the equality of the key component `Num` implies the equality of the relation elements:

```
ALL s1, s2 IN SupplierRel (s1.Num = s2.Num) => (s1 = s2)
```

An empty key component list as in `Point2DSetType` is a synonym for an enumeration of all components of the element type; in this case a relation is just a set of relation elements.

3.3 Generalized Scope and Lifetime Concepts

In a classical relational database system, a `CREATE TABLE` command implies (1) the declaration of an (anonymous) relation type, (2) the declaration of a named relation variable of this type, and (3) the specification that this relation variable should persist, i.e., preserve its state until explicitly deleted. In DBPL, we again exploit programming language principles and strive for a greater modeling power by decoupling the concepts of type, scope, and lifetime.

In order to create "containers" for values of (relation) types in DBPL, it is necessary to declare named variables explicitly:

```
VAR SupplierRel : SupplierRelType;
    OldSupplier : SupplierRelType;
    MadeFromRel : MadeFromRelType;
    GroupedSupp : GroupedSuppRelType;
```

The scope and lifetime of variables depend on the context of their declaration. Variables declared within a procedure are visible within their enclosing procedure and are created and destroyed automatically on block entry and exit, respectively. Successive or recursive procedure invocations create independent variable instances.

Similarly, variables declared in the outermost block of a program are visible within the full program and they are created and destroyed automatically on program entry and exit, respectively. Multiple program activations (sequentially or simultaneously) create independent variable instances.

The module concept makes it possible to declare named objects (types, constants, variables, procedures, ...) that are in the *scope* of multiple programs. Typically, an application program consists of a multiplicity of modules. DBPL supports separate compilation, i.e. modules can be developed independently and incrementally. A module can import names that are exported from other modules that include definitions for these names or that

```
DATABASE DEFINITION MODULE SupplierPartDB;
TYPE PartNumType        = [0..99999];
     SupplierNumType    = [1000..9999];
     SupplierStatusType = (unimportant, important, veryImportant);
     String             = ARRAY [0..29] OF CHAR;
     Dollar             = REAL;
     Kilo               = REAL;
     PartStateType      = (base, comp);
     SupplierRecType = RECORD
                Num     : SupplierNumType;
                Name    : String;
                Status  : SupplierStatusType;
               END;
     MadeFromSubRecType = RECORD
                Num     : PartNumType;
                Quantity: CARDINAL;
               END;
     MadeFromSubRelType = RELATION Num OF MadeFromSubRecType;
     PartRecType = RECORD
                Num     : PartNumType;
                Name    : String;
                CASE State : PartStateType OF
                  base  :
                    Cost        : Dollar;
                    Mass        : Kilo;
                    SuppliedBy  : SupplierNumType;
                | comp  :
                    MadeFrom    : MadeFromSubRelType;
                    AssemblyCost: Dollar;
                END;
               END;
     SupplierRelType = RELATION Num OF SupplierRecType;
     PartRelType = RELATION Num OF PartRecType;
VAR SupplierRel       : SupplierRelType;
    PartRel           : PartRelType;
    HighestPartNoUsed : PartNumType;
END SupplierPartDB;
```

Figure 1: An extended relational database schema in DBPL

```
MODULE Application;
FROM SupplierPartDB IMPORT SupplierRelType, PartRel, SupplierRel;

TRANSACTION DeleteSuppliers(Suppliers: SupplierRelType): BOOLEAN;
BEGIN IF SOME bp IN PartRel (bp.State = base) AND
         SOME s IN Suppliers (bp.SuppliedBy = s.Num) THEN
        RETURN FALSE; (* referential integrity violated *)
      ELSE
        SupplierRel:- Suppliers;
        RETURN TRUE
     END;
END DeleteSuppliers;

BEGIN ... END Application;
```

Figure 2: An application program that imports shared database objects

import these names from a third module. The compiler enforces the consistent use of names across module boundaries. Program modules have a *copy semantics* in the sense that variables visible in the scope of multiple programs are implemented by independent variable instances.

In DBPL we extended modules using a second concept which we call `DATABASE` modules (see Fig. 1). All variables declared within such a module are *persistent* and *shared*: in contrast to other program variables their *lifetime* exceeds a single program execution [29] and concurrent processes share a common variable instance. In this sense database modules have a *reference semantics*. The lifetime of a persistent variable is longer than that of any program importing it.

Using these rules it is a straightforward matter to model the scoping rules of conventional relational database systems. A database schema is simply a database module that declares and exports names for types of appropriate basic domains and declares and exports persistent variables of relation types (see Fig. 1). Similarly, an application program is a module that explicitly imports names from a database schema (see Fig. 2).

Read and write access to shared variables is required to be part of the execution of a *transaction* [30]. The ACID property [31] of DBPL transactions allows database programmers to abstract from concurrency and recovery issues since transactions are atomic with respect to their effects on persistent and shared database variables (see Fig. 2).

The potential arising from the separation between type and lifetime is exemplified in Fig. 1 by the declaration of single persistent and shared integer variable `HighestPartNoUsed` which issues unique part numbers. Another example is the transaction in Fig. 2 that uses a (temporary) parameter `Suppliers` of type `RELATION` to specify a set of suppliers to be deleted from the database variable `SupplierRel`.

Another highly relevant modeling alternative supported by DBPL (the

ADT style) is to encapsulate database variables like `SupplierRel` and `PartRel` in Fig. 1 into a DBPL *implementation module* and to restrict update operations to "trusted" transactions exported by its corresponding *definition* module [32].

Since the import relationships between modules have to be declared statically, there is no possibility for name conflicts and schema incompatibilities at run-time. Furthermore, since we allow for the definition of multiple (database) modules, the concept of *multibases* is also covered by DBPL.

3.4 Bulk Expressions and Operations

The extension of the type space available to DBPL programmers through an orthogonal bulk type constructor `RELATION OF` is complemented by an orthogonal extension of the DBPL expression language by bulk value constructors and associative value selectors. Again, we aimed for high language economy by supporting full orthogonality between Modula-2's arithmetic and Boolean expression syntax and DBPL's bulk expression syntax.

For each type constructor of DBPL (record, array, relation) there is a *value constructor* to create objects of the composite type by enumerating its components:

```
v1:= SupplierRecType{11, "John", important};
v2:= MadeFromSubRecType{11, 100};
v3:= PartRecType{3, "nut", base, 300.0, 20.3, 11};
v4:= PartNumSet{1, 3, 77};
v5:= Point2DSetType{ {1.0, 2.0}, {2.0, 1.0} };
v6:= PartRelType{ {4, "bolt", base, 30.2, 11.4, 11} };
```

The right-hand sides of these assignments make use of value constructors for record values (assigned to `v1`, `v2`, `v3`) and relation values (assigned to `v4`, `v5`, `v6`). Note that the curly brackets `{}` are *overloaded*: depending on the type identifier preceding them they construct records, arrays, or relations. The assignments to variables `v5` and `v6` contain nested value constructors, for example, `v6` is assigned a relation that contains a single record of its element type `PartRecType`. Type identifiers for nested value constructors can be omitted.

In DBPL there are three kinds of *value selectors* for the selection of components within a structured value. Elements of an array are selected by an index value of their index type, enclosed in square brackets (`vector[7]`). Fields of a record are selected by their field name (`supplier.Name`). Elements of a relation are selected by their key value, enclosed in square brackets (`SupplierRel[7]`). Therefore, variable designators of DBPL consist of a name followed by a path of value selectors:

```
SupplierRel[7].Name:= "Peter";
PartRel[3].Mass:= 13;
```

In addition to these element-wise operations, DBPL provides specialized set-oriented *bulk expressions* for relation types. There are three kinds of bulk

expressions, namely Boolean expressions, selective and constructive access expressions.

Boolean Expressions yield a truth value (`TRUE` / `FALSE`) and may be nested:

```
SOME s IN SupplierRel (s.Name = "John")

ALL s IN SupplierRel (s.Status = important)

ALL p IN PartRel (p.State <> base) OR
  SOME s IN SupplierRel (p.SuppliedBy = s.Num)
```

The comparison operators (=, <>, >=, <=, <) for relations are abbreviations for key-based quantified expressions, e.g., `SupplerRel1>=SupplierRel2` denotes relation containment and is equivalent to

```
ALL s1 IN SupplierRel1 SOME s2 IN SupplierRel2 (s1.key = s2.key)
```

The test for membership (`thisSupplier IN SupplierRel`) is equivalent to

```
SOME s IN SupplierRel (thisSupplier.key = s.key)
```

Selective Access Expressions are rules that select subrelations.

```
EACH s IN SupplierRel: s.Status = important
```

selects all elements `s` of the relation variable `SupplierRel` that fulfil the selection predicate `s.Status = important`.

A selective access expression within a relation constructor denotes a relation of all selected elements:

```
SupplierRelType{EACH s IN SupplierRel: s.Status = important}
```

Constructive Access Expressions are rules for the construction of relations based on the values of other relations:

```
NameRecType{p.Name, s.Name} OF
   EACH p IN PartRel, EACH s IN SupplierRel:
     (p.State = comp) AND (p.SuppliedBy = s.Num)
```

where `NameRecType` is a record of two strings defined as

```
TYPE NameRecType = RECORD Part, Supplier: String END
```

The construction rule above declares that pairs of names of all base parts together with their suppliers from the two stored relations `PartRel` and `SupplierRel` are to be constructed.

The application of a relation constructor to a constructive access expression creates a relation that contains the values of the target expression (preceding the keyword `OF`), evaluated for all combinations of the element variables (`p`, `s`) that fulfil the selection expression `(p.State = comp) AND (p.SuppliedBy = s.Num)`:

```
NameRelType{{p.Name, s.Name} OF
     EACH p IN PartRel, EACH s IN SupplierRel:
       (p.State = comp) AND (p.SuppliedBy = s.Num)}
```

results in a relation value which has to comply with

```
TYPE NameRelType = RELATION OF NameRecType;
```

Note that access expressions do not denote relations; only in the context of a relation constructor `RelType{...}` do they evaluate to a relation value. Other contexts in which access expressions can be used are given below.

In addition to these (side-effect free) bulk expressions, DBPL provides specialized *bulk operators* (`:=, :+, :-, :&`) for relation alterations which assign, insert, delete, and update sets of relation elements:

```
PartRel:= PartRelType{};
SupplierRel:- SupplierRelType{EACH s IN SupplierRel: s.Status=important}
```

The types of the expression and the variable on the left-hand side have to be compatible according to the rules of section 3.1. As illustrated by the examples above, the nesting of DBPL expressions captures the essence of relational query languages, namely to provide *iteration abstraction* by means of high-level set-oriented selection, construction and update mechanisms.

3.5 Exploiting Language Orthogonality

It should be noted that the database and programming language features of DBPL sketched in the previous sections do not simply stay side-by-side. On the contrary, there are many interfaces to create synergy between these features, for example:

- Relation types are allowed to appear in arbitrary contexts, i.e. not only as types for database variables, but also as types of local variables within procedures, or as types of value- or variable-parameters;
- Quantified expressions can appear not only within bulk expressions but also in conditionals or as termination conditions of loops;
- Relation constructors can be used freely within expressions of arbitrary types. A relation constructor can contain function calls, arithmetic operations etc.

Relation constructors provide an expressive mechanism to build relations from their elements. To support the inverse operation ("de-setting") there is a strong need for *iterators*. The essence of an iterator is a selective access expression (see 3.4) denoting selected elements of a relation variable to be iterated over. The body of an iteration is an arbitrary statement sequence that can read and update the value of the loop variable:

```
FOR EACH s IN SupplierRel: s.Status = important DO
  InOut.WriteString(s.Name); InOut.WriteLn;
END;
```

Up-to-date programming languages provide two important abstraction mechanisms to achieve localization of information in large software systems [24, 33]. Process abstraction allows programmers to abstract from the implementation of a subroutine and to perform complex operations simply through reference to its name with an appropriate list of actual parameters. Type abstraction allows programmers to abstract from the implementation of a data structure and to operate on it only via a well-defined interface expressed as a set of operations defined for an abstract data type.

DBPL embodies both abstraction mechanisms by means of procedures that abstract over statements, functions that abstract over expressions, and opaque types that abstract over type expressions [10]. In addition, DBPL provides *selectors* that abstract over selective access expressions and *constructors* that abstract over constructive access expressions (see 3.4) [34]. These two abstractions capture the essence of updatable and non-updatable *views* in relational databases since selector applications can appear wherever a relation variable is expected and constructor applications can appear wherever a relation expression is expected.

The selector `SuppliersWithStatus(important)` defines an *updatable* view on the supplier relation selecting those suppliers having `important` as their status (see also Sec. 3.4):

```
SELECTOR SuppliersWithStatus(st: SupplierStatusType): SupplierRelType;
BEGIN EACH s IN SupplierRel: s.Status = st END SuppliersWithStatus;
```

The following constructor `SuppliersForParts` names a *non-updatable* view that is derived from the base relations `PartRel` and `SupplierRel` and that contains pairs of parts and supplier names for all base parts with their respective suppliers.

```
CONSTRUCTOR SuppliersForParts: NameRelType;
BEGIN
  NameRecType{p.Name, s.Name} OF
     EACH p IN PartRel, EACH s IN SupplierRel:
       (p.State = comp) AND (p.SuppliedBy = s.Num)
END SuppliersForParts;
```

Without going into details it should be noted that naming (of statements, expressions etc.) naturally leads to the concept of recursion. The semantics of a recursive query expression in DBPL is not defined operationally (as is common practice for procedures) but as a least fixed point of recursive set equation [19, 34]. Thereby, constructors are at least as expressive as recursive DATALOG programs with stratification semantics [35, 36] over complex objects:

```
TYPE NumPairType = RELATION OF RECORD Super, Sub : PartNumType END;

CONSTRUCTOR TransitiveMadeFrom: NumPairType;
BEGIN
  {p.Name, mf.Num} OF
     EACH p IN PartRel, EACH mf IN p.MadeFrom:
       p.State = comp,
  {mf1.Super, mf2.Sub} OF
    EACH mf1 IN {TransitiveMadeFrom}, EACH mf2 IN {TransitiveMadeFrom}:
      mf1.Sub = mf2.Super
END TransitiveMadeFrom;
```

4 DBPL System and Environment

The development of data-intensive and long-lived applications is recognized as a demanding system construction task that requires full software-engineering support. In implementing DBPL we therefore paid attention to smoothly integrate the DBPL tools into the native software development environments available on DBPL's host architectures. In several DBPL spin-off projects we also studied more advanced issues of open distribution, commercial interoperability and computer-aided software engineering (see Sec. 5.2).

We based the DBPL system implementation on state-of-the-art relational database and compiler technology. We were, however, compelled to invest a significant amount of development work to overcome mismatches of the contributing technologies that otherwise would have disrupted the clean and simple DBPL language model described in the previous section. In section 4.2 and 4.3 we therefore report on experience gained during the past six years of DBPL system (re-)implementation and refinement in

- removing conventional compiler implementation restrictions,
- extending the expressiveness of the bulk query language,
- generalizing the storage model, and
- reducing system performance bottlenecks resulting from mismatches between independently developed contributing technologies.

In section 4.4 we share some of the conclusions we have drawn from the DBPL implementation effort for our future work on database programming language implementations.

4.1 Architectural Overview

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

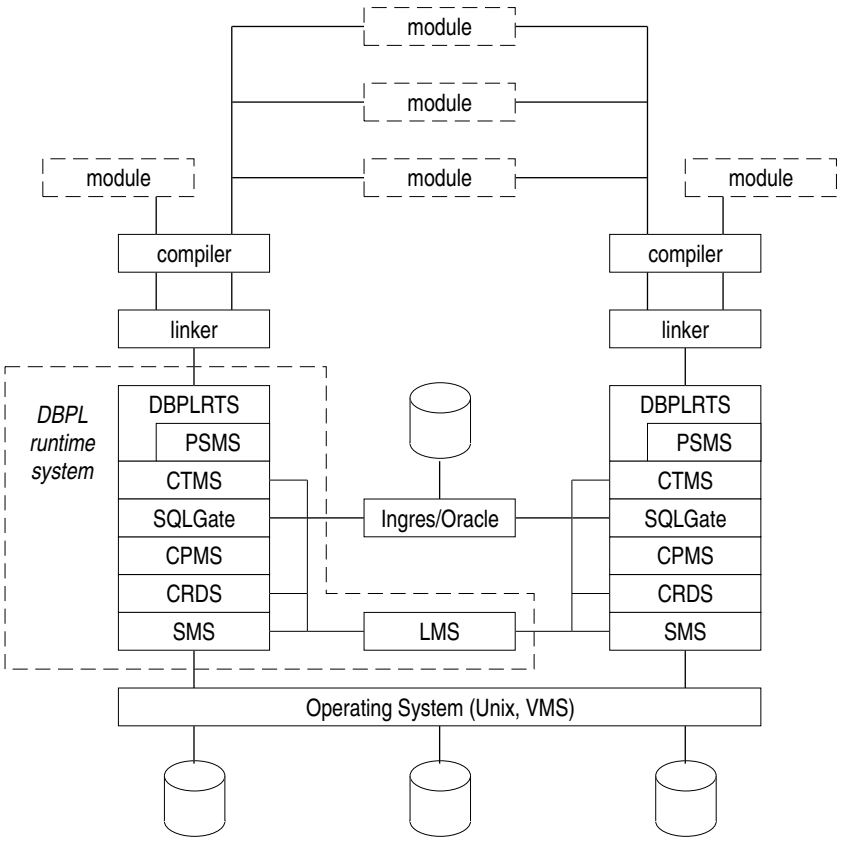

Figure 3: Overall DBPL system architecture

application link-time, while the run time system looks after dynamic aspects of database applications such as query optimization, storage management, serialization of transactions and failure recovery for persistent data.

Figure 3 depicts the overall DBPL system architecture as well as 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. 3). 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 run-time, this executable program interacts through the interface module `DBPLRTS` with the various layers of the DBPL run time system. 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 dis-

tinction between DBPL and SQL database objects is fully transparent to the application programmer.

Fig. 3 also shows 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 SQL servers. Due to limitations of the currently available SQL server interfaces there is no two-phase commit protocol to guarantee abort and commit consistency across mixed DBPL and SQL databases in case of system failures at commit-time.

Currently, we are distributing DBPL in two distinct system implementations, VAX/VMS DBPL and Sun DBPL. Application programs developed with one system (including user-interface code) can be recompiled without change using the other system. Both systems are written entirely in Modula-2 and consist of a compilation and a run-time environment. The multi-user DBPL 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 fully into commercially available software development environments (NSE, Teamware, CMS) and provide an optional SQL gateway in addition to the DBPL-specific storage manager.

4.2 Extending Compilation Technology

By recasting the concept of schemata into the concept of shared modules, we were able to successfully exploit the compilation and module management technology developed for modular languages like Ada, Modula-2, Modula-3 and Eiffel. We extended this technology by link-time mechanisms to check the consistency of persistent variables against the type information used by their importing applications. Furthermore, we developed "binding tools" to support simple persistent data conversions as required by typical evolutionary schema changes. These tools also allow database administrators to bind dynamically persistent variables to alternative type-compatible implementations like internal DBPL storage structures or external (Ingres, Oracle) SQL database relations.

The scoping and polymorphic type checking required at compile-time for relational expressions and (higher-order) selectors and constructors was implemented (analogously to array operations and procedures) by hand-coded checking routines, one for each language construct. The DBPL compiler also performs a limited form of type inference to eliminate the need for "redundant" type annotations in nested queries. While these tasks are captured adequately in the larger framework of polymorphic type systems (like ML [37]

or Quest [38]), the DBPL system also calls for *dynamic* type descriptions available at run-time to perform, for example, type-dependent tuple value equality comparisons.

The interaction between imperative programming language statements and declarative, dynamically optimized bulk operations in DBPL is solved technically as follows: DBPL operations on individual relation elements and navigational operations are mapped by the compiler directly to operations of the run-time system. Bulk 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 of this, the run-time system receives a compact internal representation of the operation 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 run-time system, thereby binding operands to program variables.

3. Optimization and evaluation of the operations on the dynamically bound operands as defined by the predicate tree.

Selectors and constructors are represented at run-time 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 run-time system provides symbolic operations for the manipulation of predicate trees (copy, delete, expand, ...). 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.

It was possible to undertake virtually all extensions from Modula-2 to DBPL without interference with the machine dependent parts of the code generation and to obtain a high portability of the compiler front end. Many operations of DBPL are (conceptually) implemented by sequences of equivalent statements of Modula-2, containing calls to the run-time support module `DBPLRTS` (see Fig. 3). Accordingly, the VAX DBPL and the Sun DBPL compiler utilize the same compilation strategies despite significant differences in the details of the target code generation.

4.3 Extending Database Technology

The architecture of the DBPL run-time system is similar to the layered architecture of other extended or extensible relational database systems [39, 40]. As can be seen in Fig 3, the DBPL run-time support subsumes a storage management system (SMS) for fixed-sized records and variable-sized long fields

(maximum 8MB in size), a complex-object relational data system (CRDS), a predicate management system for the optimization and evaluation of composite relation- and boolean-valued expressions (CPMS), a database deduction engine to perform optimized fixed-point algorithms over sets of predicates (PSMS), a predicative set-oriented transaction-manager (CTMS, see also [25]), and an ADT-based interfaced (DBPLRTS) to the DBPL compiler and other DBPL tools.

A repeating pattern in the development of the DBPL system was the need to generalize existing database solutions to meet the requirements imposed by the orthogonal DBPL language model. At the storage level, for example, we had to substantially revise the interface to the "relational data storage system" to be able to identify, create, manipulate and destroy non-persistent relations, non-set-valued persistent objects, non-homogeneous collections, or to navigate through nested complex objects. Similarly, the query evaluation and optimization component had to be extended substantially to handle parameterized query expressions, non-set-valued queries, type-complete data structures, user-defined functions and arbitrarily nested query expressions.

As a result of these generalizations, the operations at the interfaces of the DBPL system layers are centered around general *programming language abstractions* like value, variable, type, scope, object identifier, parameter and expression and no longer around DBMS-specific combinations of these concepts like tuple, relation, dictionary, TID, view, or query tree.

4.4 On the DBPL System Complexity

Figure 4 gives a rough quantitative overview of the relative complexity of the different tasks in the DBPL system. The complexity is distributed fairly equally between the run-time system, the compiler front-end that performs type checking, inter-module consistency checking, intermediate code generation and the industrial-strength compiler back-end that performs object code generation and optional intermediate 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 compactness and relative efficiency of the DBPL run-time system results from the fact that a large number of scanning, parsing, type checking, error recovery and consistency checking tasks performed by conventional database systems was factored-out into the DBPL compiler. In particular, there is no need to check the parameters or the nesting of run-time system calls (e.g., hidden cursor operations, buffer creation and deallocation) since these calls are generated exclusively by the compiler and not by application programmers.

Fig. 4 also reveals a disadvantage of the multi-level concurrency control and recovery scheme adopted in the DBPL system [41]. The layer CRDS that is responsible for the mapping of tuple-level complex object operations to operations on atomic fragments maintained by the storage manager SMS does

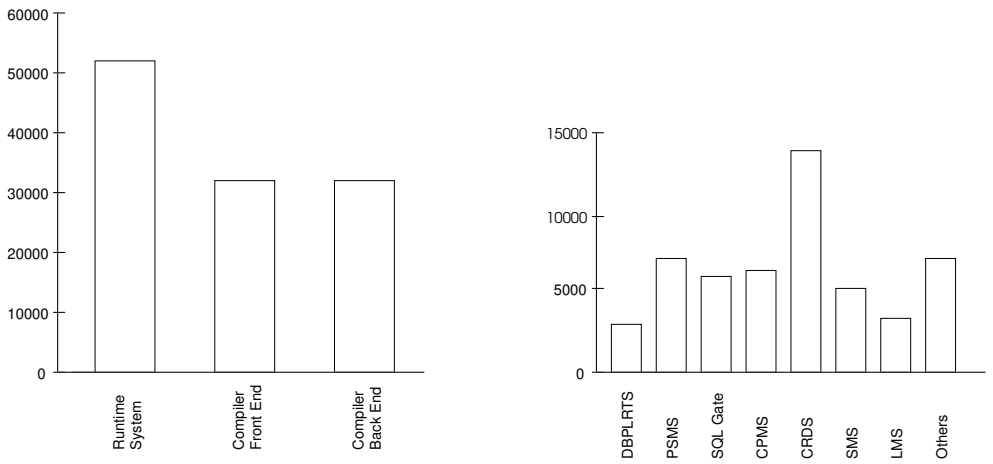

Figure 4: Relative sizes of DBPL system components (lines of Modula-2 code)

not rely exclusively on concurrency control and recovery primitives of the SMS but performs its own concurrency control and recovery mechanisms exploiting the semantics of complex object manipulations. As can be seen in Fig. 4, the price for a gain in parallelism is a significant increase in the complexity of this system component. Furthermore, bugs in the recovery code are very hard to locate since they only become visible if the system is operated in stress situations. In retrospect, we feel that we should have chosen a less aggressive approach to scheduling and fault tolerance in DBPL.

The heavy dependency of the DBPL compilers for Sparc, VAX, Motorola and Intel architectures on architecture-specific object code, debugger information and link formats turned out to be a major issue in the long-term maintenance of the DBPL system. In our current research projects outlined in Sec. 6 we therefore moved to platform-independent, optimizable and *persistent* code representations based on continuations [42]; a move that seems to give rise to interesting new database programming concepts like run-time reflection, persistent threads, or the dynamic distribution of data and code across architecture boundaries [17].

The Open DBPL/SQL gateway (compare Sec. 5.2.2) and the bidirectional call interfaces between DBPL and other languages demonstrated that it requires a surprisingly small effort to embed a "closed" database programming language with a well-designed internal architecture into "open" system environments. On the other hand, this effort was highly appreciated by DBPL users since these generic gateways enable the construction of hybrid application systems that make best use of the technologies from both worlds.

5 A Short Reflection on DBPL's Development

DBPL was not designed in "one shot" but developed in phases which reflect rather clearly our degree of understanding and our level of ambition. In the

mid-70ies we were not the only group working on algorithmically complete languages for data-intensive applications — in the relational setting the correspondence between central concepts in programming and database modeling seemed obvious and it was agreed that types were the notion to bridge the gap [43, 44, 45]. However, it took us more than 30 man-years to develop the concepts underlying a relational database programming language into the DBPL system which fully exploits them and provides sufficient efficiency and operational support for their effective use.

On the other hand, DBPL was not the end of our efforts in supporting database programming. We embedded it into a network of DBPL-based spin-off projects ranging from distributed DBPL [20] to SQL server gateways [13] and DBPL-oriented programming environments [46].

5.1 DBPL as a Learning Experience

In Pascal/R [1] we concentrated on capturing and integrating relational semantics in the context of a strongly and statically typed monomorphic programming language. We introduced relation types to model schemata, relation expressions for queries and generalized assignment statements for relation updates. A database was an aggregate of relation variables imported through external program variables in the same manner as Pascal works with external files. Optimization of relation expressions was another large issue at that time [47] and some of the algorithms designed by Matthias Jarke and Jürgen Koch [18] still seem to be alive in DBPL. Since nesting of relational expressions was heavily used for controlling subquery execution, we soon discovered that something must be wrong with the standard predicate transformation rules in cases of empty range relations: universal quantifiers do not necessarily commute in this case. Frank Palermo visited us in Hamburg and helped us by fixing his algorithms [48]. Another issue was that of update semantics in the case of key constraint violations. Keys while being part of type definitions destroyed static type checking for relations. Pascal/R introduced specific run-time exceptions on relation updates, an experience which motivated our subsequent emphasis on transactions and on exception handling in general.

The Pascal/R system was used quite extensively and successfully in education including about 20 installations abroad. However, as soon as we started implementing bigger database applications with Pascal/R (e.g., FIBEX [49]) we quickly experienced two rather severe deficiencies. The first inadequacy was the missing transactional support in Pascal/R, the second one was a general deficit inherited from Pascal, namely its lack of support for modular system programming. We recognized that, in order to make progress with system programming support, we had to leave the framework of Pascal. Before initiating the DBPL project we began an intensive period of concept development and evaluation in order to improve on the deficits mentioned above[4]. As a consequence, we decided to move to Modula-2 as a host and implementation

[4]This work was supported by DFG grant Schm.450/2-1 ("Datenbankprogrammiersprachen").

language, an effort we started by porting the ETH/Zurich compiler into the DEC/VMS environment [26] (As a side effect we still hold more than 600 licenses for that product!).

We started working on transactional abstraction, work leading to parameterized transaction procedures [30]. Furthermore, we recognized the central importance of iteration abstraction for relation value construction, relation iteration control and relation view definition (see Sec. 3.5). Selectors were introduced as named and parameterized access expressions thereby supporting iteration abstraction by a first-class concept in database languages.

In 1980 we began a close cooperation with ETH/Zurich on Modular/R [50] (by having Manuel Reimer from Hamburg working at ETH). This small project was essentially an exercise in evaluating the concepts developed for iteration and transaction abstraction. Furthermore, we started exploiting Modula's module concept for database programming. Having modules available as a general concept to organize code sharing between programs surely influenced our next step to exploit modules also for data sharing [51]. We introduced persistent database modules which, in Modula/R, were still restricted to relations only. Relation types and operations as well as the work on query optimization were carried over in a fairly straightforward way from Pascal/R into Modula/R and then further into DBPL.

Based on the above experience we decided around 1983 to start the DBPL project. Our vision for DBPL was as simple as ambitious [52]: define and implement a language which provides, in an algorithmically complete context, its four database concepts – relation type, access expression, database module, transaction procedure – *unrestrictedly*.

During DBPL development it became obvious that both communities, databases and programming languages, were adhering to several, mostly unnecessary and ad-hoc restrictions. Some limitations, for example, the lack of data *type orthogonality* (i.e. the restriction to first normal form relations), had already been realized, others, such as *orthogonal persistence* (providing persistence for all types, not just for relations), were new for the database community (but already discovered by the persistent programming language people [29]). Other insights, for example, the relationship between abstract access expressions and queries, iterators and views, or between recursive access expressions and deductive databases, came as a surprise, at least to us.

5.2 DBPL Spin-Off Projects

DBPL initiated several spin-off projects, all of which were triggered by the fact that a language is hardly ever used in isolation but has to enable good-quality interaction in heterogeneous environments consisting of other programs, databases, tools, machines etc. Therefore, we embedded DBPL as a product deeply into its native environments (see Sec. 4.1) and we used DBPL as a research vehicle for several R&D projects to investigate issues of distribution, interoperability and computer-aided software engineering.

5.2.1 DURESS: Distributed DBPL in Heterogeneous Environments

In the DURESS project[5] [20, 53] we exploited DBPL's module concepts described in section 3.3 to add an additional layer of type-safety to standard remote procedure call mechanisms (RPC) in federated client-server programming environment. In DURESS, individual DBPL compilation units may be located at different nodes in a (LAN/WAN) network consisting of PC workstations and VAX servers.

In the distributed DBPL environment, the compiler automatically generates the necessary client and server stub code to access procedures exported from remote sites. Furthermore, dynamic type descriptions are assigned to compiled remote interface definitions. These descriptions are used during connection establishment to verify that all parts of a distributed program have been compiled using compatible versions of the interface specifications. This mechanism directly generalizes the link-time consistency control mechanisms developed for persistent DBPL modules.

In DURESS, remote modules can also export (database) variables, transactions and views (represented as selectors and constructors). In this case, the extended DBPL run-time system performs the necessary low-level communication and distributed query evaluation to blur the distinction between local and remote objects:

```
DATABASE DEFINITION RemoteSupplierDB ON "Server7";
IMPORT SupplierPartDB;
VAR SupplierTable : SupplierRelType;
END RemoteSupplierDB;
```

5.2.2 Open DBPL/SQL: A Gateway to Commercial SQL Servers

In another subproject[6] we studied issues of DBPL system interoperability by extending the DBPL system to support bindings to external persistent objects in addition to internal persistent DBPL objects [13]. The only change to the language definition of DBPL is the syntax of definition module headers:

```
DATABASE DEFINITION FOR Ingres MODULE Suppliers2;
IMPORT SupplierPartDB;
VAR SupplierTable : SupplierRelType;
END Suppliers2;
```

This module exports a persistent variable `SupplierTable` that is implemented by the table `SupplierTable` stored in the Ingres database `Suppliers2`. All DBPL statements and expressions referring to such an "external" variable are translated fully transparently into SQL update and selection expressions submitted to the Ingres SQL database management system. These SQL expressions typically take DBPL program variables as arguments and return (relation) values that are converted appropriately for further processing within DBPL. For example, the query

[5]DURESS was supported by a grant from the IBM European Network Center, Heidelberg.
[6]This work was supported by a grant of the DAAD, Bonn.

```
IF ALL s IN SupplierRel (s.Status > x) THEN ... END
```

is translated into a `select from where` SQL expression that uses the actual value stored in the DBPL variable `x` of type `SupplierStatusType`. Depending on the cardinality of the set-valued result, a boolean value is then returned to the compiled DBPL code.

It should be noted that DBPL can handle arbitrarily nested (possibly recursive) query expressions that mix volatile relations, persistent DBPL relations residing in local or remote databases and SQL database relations. Therefore, much care has been devoted to develop evaluation heuristics that minimize data transfer and make best use of index information available for individual relations. Evaluation strategies are not determined at compile-time but depend on cardinality and index information available at run-time.

5.2.3 DAIDA: An Environment for Data-Intensive Application Development

In the DAIDA project[7] [54, 55, 46], software houses, research centers and universities from five European countries have contributed to solving the problem of intelligent development assistance for database applications. Within the integrated DAIDA framework, DBPL served as the target language for the implementation of high-quality database application software while the application requirement analysis task and the conceptual modeling process are expressed using specialized high-level languages (Telos and Taxis DL, respectively). A central contribution of DAIDA is the development of methodologies and tools that assist the system analyst, system designer and database programmer in the mapping process between the three languages involved. The mappings and the associated design decisions are recorded in a global knowledge base (ConceptBase [56]) to support the long-term evolution and maintenance of software systems.

6 On Project Termination

To summarize, the most prominent feature of the DBPL language is the type-complete integration of sets and first-order predicates into a strongly typed, monomorphic language with persistence as an orthogonal property of individual compilation units. The DBPL system is further characterized by its coverage of a wide range of operational database demands, such as query optimization, transaction and distribution management and computer-aided support for database application development.

Over the years, we encountered several data modeling concepts (inheritance, object identifiers, ordered bulk data structures, triggers, ...) and database system requirements (communication between concurrent processes, exception handling, ...) that did not fit nicely into the original DBPL language framework. However, instead of "fixing" these DBPL limitations by

[7]DAIDA was supported by Esprit Project #892.

ad-hoc, incremental language extensions, we tried to isolate the conflicting system or model assumptions in the initial DBPL (or Modula-2) design.

More specifically, we regard the monomorphic type system of Modula-2, the lack of first-class procedures, and the strict separation between built-in and user-defined system functionality in the DBPL system architecture as hard, built-in limitations which prevent, without violating our design principles, DBPL extensions beyond the realms of set- and predicate-based data models and first-order programming languages.

As a consequence, our work in the Tycoon project[8] [57] is based on a higher-order polymorphic core language consisting of a small set of highly generalized primitives for naming, typing and binding [58]. Our initial Tycoon experiments [15, 59, 9] suggest that the Tycoon framework by virtue of the orthogonality of its contributing concepts goes significantly beyond conventional relational, object-oriented and deductive approaches.

In a similar vein, the Tycoon system architecture [17] aims at increased system scalability, portability and interoperability with commercial and standardized systems by "unbundling" database functionality and separating data storage, data manipulation and data modeling. To summarize, the Tycoon project carries forward in a more general linguistic and architectural framework the experience gained with bulk data typing, iteration abstraction, and modularization in DBPL.

[8]Tycoon is supported in part by Esprit project #6309 (FIDE$_2$).

References

[1] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977. (Also appeared in ACM TODS, 2(3), September, 1977 and A. Wasserman (editor), IEEE Tutorial on Programming Language Design, and M. Stonebreaker (editor), Readings in Database Systems, Morgan Kaufmann Publishers, 1988 and 1993).

[2] J.W. Schmidt. Type Concepts for Database Definition. In B. Shneiderman, editor, *Databases: Improving Usability and Responsiveness*. Academic Press, 1978.

[3] J.W. Schmidt and M. Mall. Pascal/R Report. Bericht 66, Fachbereich Informatik, Universität Hamburg, Germany, January 1980.

[4] J.W. Schmidt and F. Matthes. Modular and Rule-Based Database Programming in DBPL. In M. Jarke, editor, *Database Application Engineering with DAIDA*, volume 1 of *Research Reports ESPRIT*, pages 85–122. Springer-Verlag, 1993.

[5] F. Matthes and J.W. Schmidt. DBPL: The System and its Environment. In M. Jarke, editor, *Database Application Engineering with DAIDA*, volume 1 of *Research Reports ESPRIT*, pages 319–348. Springer-Verlag, 1993.

[6] F. Matthes and J.W. Schmidt. Datenbankprogrammiersprachen. *Informatik Spektrum*, 15(4), August 1992.

[7] F. Matthes, A. Rudloff, J.W. Schmidt, and K. Subieta. The Database Programming Language DBPL: User and System Manual. FIDE Technical Report Series FIDE/92/47, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, July 1992.

[8] F. Matthes and J.W. Schmidt. The Type System of DBPL. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 255–260, Salishan, Oregon, June 1989.

[9] F. Matthes and J.W. Schmidt. Bulk Types: Built-In or Add-On? In *Proceedings of the Third International Workshop on Database Programming Languages*, Nafplion, Greece, September 1991. Morgan Kaufmann Publishers.

[10] N. Wirth. Report on the Programming Language Modula-2. In *Programming in Modula-2*. Springer-Verlag, 3rd edition, 1985.

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

[12] H. Eckhardt, J. Edelmann, J. Koch, M. Mall, and J.W. Schmidt. Draft Report on the Database Programming Language DBPL. DBPL-Memo 091-85, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1985.

[13] F. Matthes, A. Rudloff, J.W. Schmidt, and K. Subieta. A Gateway from DBPL to Ingres: Modula-2, DBPL, SQL+C, Ingres. FIDE Technical Report Series FIDE/92/54, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1992.

[14] J.W. Schmidt and F. Matthes. The Rationale behind DBPL. In *3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems*, volume 495 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1991.

[15] 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.

[16] 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.

[17] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993. (In German.).

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pages 316–325, June 1984.

[23] M.P. Atkinson and P. Bunemann. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2), June 1987.

[24] R.W. Sebesta. *Concepts of Programming Languages*. Benjamin/Cummings Series in Computer Science. Benjamin/Cummings Publishing Company, Inc., 1989.

[25] 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.

[26] 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.).

[27] F. Matthes, G. Schröder, and J.W. Schmidt. VAX Modula-2 User's Guide; VAX DBPL User's Guide. DBPL Memo 121-91, Fachbereich Informatik, Universität Hamburg, Germany, December 1991.

[28] S. Abiteboul, P.C. Fischer, and H.J. Schek. *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[29] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4), November 1983.

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

[31] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 1993.

[32] J.W. Schmidt and F. Matthes. Modular and Rule-Based Database Programming in DBPL. In M. Jarke, editor, *Database Application Engineering with DAIDA*, Research Reports ESPRIT, pages 85–124. Springer-Verlag, 1993.

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

[34] 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.

[35] J. Minker. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, 1988.

[36] 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.

[37] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[38] L. Cardelli. Typeful Programming. Report 45, Digital Equipment Corporation, Systems Research Center, May 1989.

[39] M. Carey, editor. *Database Engineering, Special Issue on Extensible Database Systems*, volume 10, June 1987.

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

[41] 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.

[42] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[43] J.E. Shopiro. Theseus – A programming language for relational databases. *ACM Transactions on Database Systems*, 4(4):493–517, December 1979.

[44] A.L. Wasserman. The Data Management Facilities of PLAIN. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 60–70, 1979.

[45] L. Rowe and K. Shoens. Data Abstraction, Views and Updates in RIGEL. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 71–81, 1979.

[46] M. Jarke. *Database Application Engineering with DAIDA*. Research Reports ESPRIT. Springer-Verlag, 1993.

[47] M. Jarke and J.W. Schmidt. Query Processing Strategies in the Pascal/R Relational Database Management System. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Orlando, FL, 1982.

[48] F.P. Palermo. A Data Base Search Problem. In *Proceedings 4th Computer and Information Science Symposium*, pages 67–101, Miami Beach, 1972.

[49] D.L. Cram, J.C. Freytag, I. Hampton, M. Mall, J.W. Schmidt, and T. Schwinghammer. The Evaluation of Acoustics Survey Data from the First BIOMASS Experiment – Report on an Interdisciplinary Data Management Project. Bericht 87, Fachbereich Informatik, Universität Hamburg, Germany, March 1982.

[50] J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, and C.A. Zehnder. Modula/R Report, Lilith Version. Technical report, Department Informatik, ETH Zürich, Switzerland, February 1983.

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

[52] J.W. Schmidt and M. Mall. Abstraction Mechanisms for Database Programming. In *Proc. SIGPLAN Symp. on Programming Language Issues in Software Systems*, San Francisco, June 1983.

[53] W. Lamersdorf, H. Eckhardt, W. Effelsberg, K. Reinhard, and J.W. Schmidt. Database Programming for Distributed Office Systems. In *Proc. Int. Conf. on Office Automation*, Washington, 1987.

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

[55] M. Jeusfeld, M. Mertikas, I. Wetzel, Jarke. M., and J.W. Schmidt. Database Application Development as an Object Modelling Activity. In *Proc. 16th VLDB Conference*, Brisbane, Australia, August 1990.

[56] M. Jeusfeld, T. Rose, and M. Jarke. ConceptBase: A Telos-Based Software Information System. In M. Jarke, editor, *Database Application Engineering with DAIDA*, Research Reports ESPRIT, pages 367–388. Springer-Verlag, 1993.

[57] F. Matthes and J.W. Schmidt. Definition of the Tycoon Language TL – A Preliminary Report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[58] J.W. Schmidt and F. Matthes. Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems. In *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering, Interoperability in Multidatabase Systems*, pages 2–16, Vienna, Austria, April 1993.

[59] F. Matthes and J.W. Schmidt. Towards Database Application Systems: Types, Kinds and Other Open Invitations. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991. (Also appeared as TR FIDE/91/14).