DBPL: The System and its Environment

Florian Matthes Joachim W. Schmidt

University of Hamburg
Department of Computer Science
Schlüterstraße 70
D-2000 Hamburg 13, FRG
matthes,schmidt@dbis1.informatik.uni-hamburg.de

Abstract

The DBPL system and its environment support the modular implementation of advanced data-intensive applications based on integrated database and programming language technology. It provides state-of-the-art system support at application compile-, link- and run-time as well as tools for incremental application evolution.

1 System Support for Data-Intensive Applications

Databases have proven to be the keystones for the majority of application systems with a wider functionality, utilization and availability. Therefore, the development of integrated language and system support for the definition, execution and maintenance of data-intensive applications gained substantial interest over the past years [?, ?, ?, ?, ?]. Essentially, this support concentrates on three major goals:

- improved data integrity;
- enhanced program efficiency;
- increased user productivity.

As expected, substantial progress towards these goals requires contributions from several key areas of computer science, in particular from

- language and compiler technology;
- optimization strategies in searching, computing and scheduling; and
- formal specification schemes for relevant properties of data and programs.

The database programming language project DBPL approaches the above goals essentially by the

- linguistic quality of the DBPL interfaces for data definition and manipulation;

[1] This work was supported by the European Commission under ESPRIT contract # 892 (DAIDA).

- distinctiveness of the DBPL system implementation; and
- richness of the DBPL environment.

It turns out to be crucial for data-intensive applications to distribute the supported tasks appropriately over *time*. While application *design time* profits from the availability of powerful abstraction mechanisms for data and transaction definition, *compile time* support is essential for (partial) correctness and consistency, proper error reporting, interface checking and certain classes of optimizations. *Run time* support is vital for global efficiency within and among transactions as well as for long-term data integrity. Finally, the extremely long *lifetime* of most data-intensive applications requires permanent support for incremental data and program modification and partial redesign.

While the DBPL language is discussed elsewhere (see, e.g. [?, ?, ?] and the chapter on "Modular and Rule-Base Database Programming in DBPL" in this volume), this paper concentrates in section 2 on the presentation of DBPL's compile time support and in section 3 on the DBPL run time system. Section 4 describes how advanced incremental compilation technology can be exploited for long-term program and system evolution in a tightly coupled interactive programming environment. Further details of the system components are described in DBPL and DAIDA technical reports (e.g. [?, ?, ?]). The paper closes with an outlook on current research in next-generation DBPL environments.

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 run time system is highly portable and runs under VAX/VMS 6.1, SunOS 4.1, IBM AIX 3.1 and IBM OS/2. All operating system dependend code is factored out into four modules for main memory management, block-oriented file input and output, exception handling and error message handling. The DBPL compilers generate native code for VAX, respectively Sparc, Motorola and Intel architectures. Both DBPL systems can be integrated deeply into the DAIDA environment and into commercially available software development environments (NSE [?], SCCS, CMS).

Figure ?? depicts the overall DBPL system architecture. The left hand side of Fig. ?? sketches how a DBPL program module is first translated by the DBPL compiler then linked with other compiled DBPL modules yielding executable object code. This object code interacts through the interface module *DBPLRTS* with the various layers of the DBPL run time system (*PSMS*, *CTMS*, *CPMS*, *CRDS*, *SMS*; see Sec. ??). Database objects are stored in files accessed through operating system calls issued by the layer *SMS*.

Fig. ?? also outlines the interaction between two DBPL applications running against shared databases. This sharing is achieved by importing the same database module(s) into two different programs. Concurrent access to database objects requires a synchronization that is achieved by explicit message passing at three layers of the DBPL system (*CTMS*, *CRDS*, *SMS*) via *LMS* services. A centralized scheduling process guarantees serializable transaction execution for DBPL applications, possibly running on different network nodes in a DECnet or TCP/IP local area network.

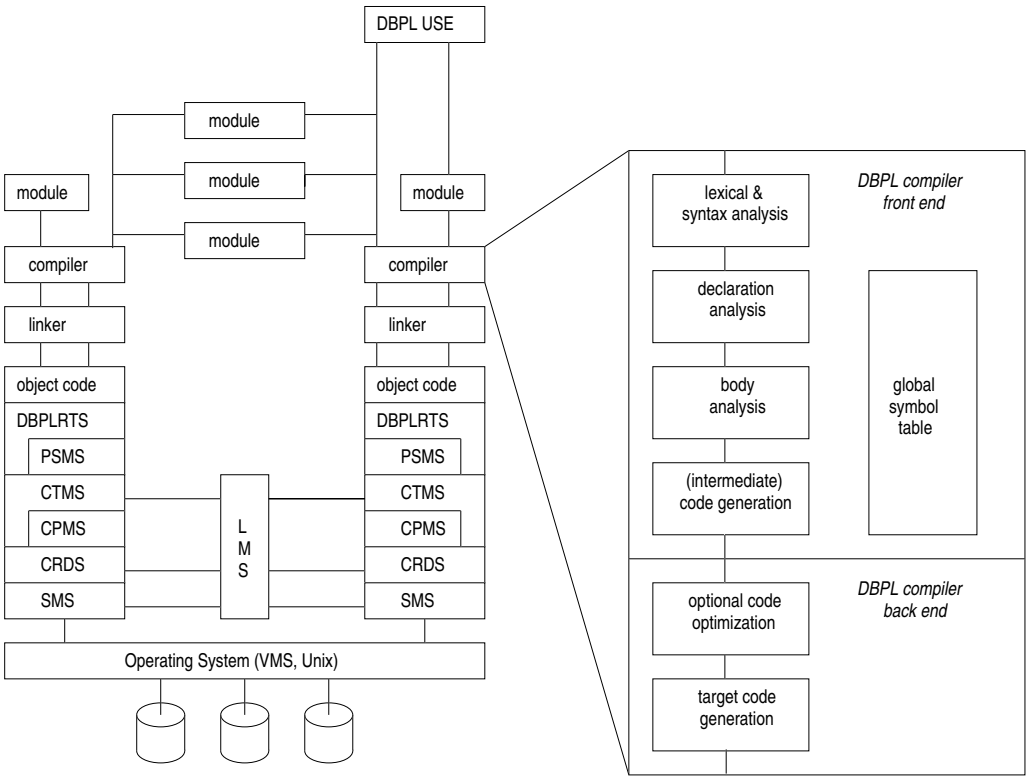

Figure 1: Overall DBPL system architecture

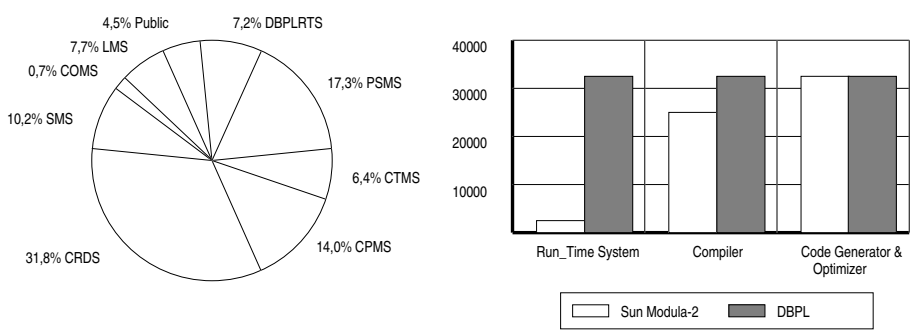

Figure 2: Relative DBPL system component sizes

2 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 [?, ?, ?].

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.

2.1 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.

2.2 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. ??). The VAX DBPL compiler directly creates relocatable object code 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 [?].

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. ??).

2.3 Overall Compiler Structure

The DBPL compiler front end (see Fig. ??) 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. ??) 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.

2.4 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.

2.5 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 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 ??). 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.

2.5.1 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.

2.5.2 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 ??. 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.).

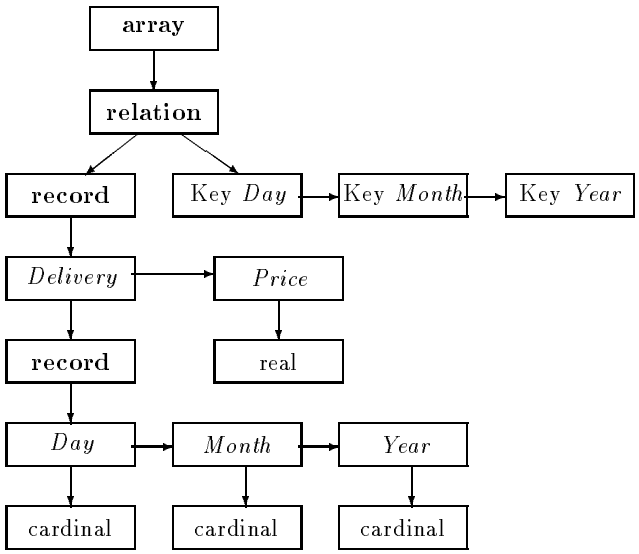

Figure 3: Run-time type descriptions

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.

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.5.6 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. ??) 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.

2.5.7 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. ??, p. ??) contains, if necessary, additional references to type structures as in Fig. ??. The generation of a predicate is performed node by node in a preorder traversal.

2.5.8 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. ??). 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.

2.5.9 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.

3 The Multi-User DBPL Database System

The layered architecture of the DBPL run time system (Fig. ??) is roughly equivalent to the architecture of other relational or object-oriented database systems [?, ?, ?, ?]. It is excelled by its support for non-relational persistent objects, temporary relations, recursive queries, complex objects and client-server architectures. Fig. ?? gives an idea of the relative complexity of the various DBPL system implementation tasks measured in lines of Modula-2 source code.

3.1 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. ??), *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 ??);
- 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. ??);
- 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 ??) 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 ??).

3.2 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 [?].

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 by *CPMS* routines (*Evaluate, BooleanValue, Assign, Insert, Delete, Update*). 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 [?], *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* [?] 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 [?] 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 evalau-

tion functions exported by the layer *CPMS*. 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*).

3.3 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. ??). 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 [?, ?]. 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 [?]) 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 [?], while the index management in the layer *CRDS* employs a tailored graph locking protocol for B-link trees [?] that supports concurrent updates. Page and record operations are synchronized using a standard strict two-phase locking scheme.

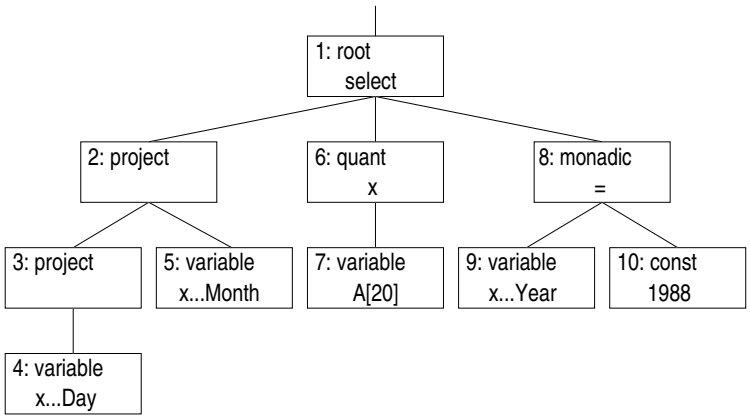

Figure 4: Predicate tree for a DBPL query expression

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.

3.4 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. ?? 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

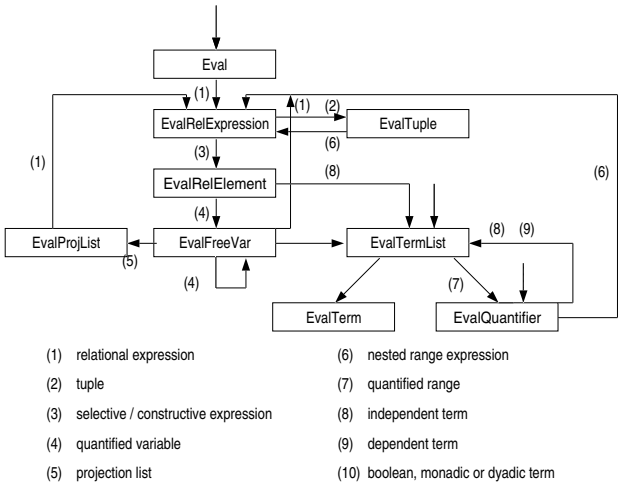

Figure 5: Evaluation procedures for DBPL queries and their mutual dependencies

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. 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 subre-

lations 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 [?, ?, ?].

Fig. ?? 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 understood 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*).

3.5 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.

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

Figure 6: Structure of a map

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 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 [?]. 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 [?] 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. ??). 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;

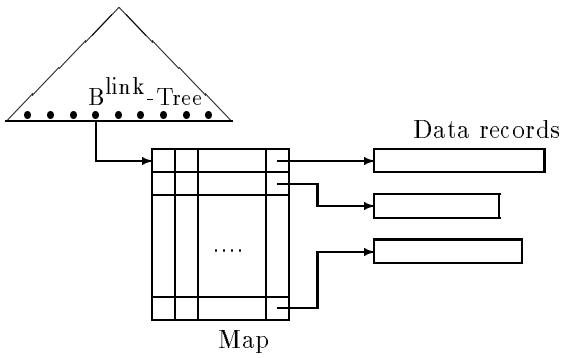

Figure 7: *CRDS* access path management

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 [Map-TID, MapIndex], elements of nested relations are addressed indirectly, this *SuperTID* represents a stable database address.

As illustrated in Fig. **??**, 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.

3.6 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 $\frac{|\text{page}|^2}{|\text{TID}|+|\text{Length}|}$. 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 execut-

ing 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. ??).

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.

4 DBPL-USE – A Language-Sensitive Editor

Data-intensive applications are developped in contexts, in which certain of the required computational objects may preexists. This applies to newly developped application programs which have to comply with already existing databases, their type definitions, predefined transactions, views etc. as well as to local modification of existing application programs which have to maintain the interfaces to the constant part of their environment.

In this situation, substantial support can be gained by editing facilities with a certain amount of syntactic and semantic knowledge about the context in which local program extension and modification takes place. This support requires, of course, an editor which is sensitive to the language in which databases are defined and applications are programmed. In an integrated linguistic framework as given by DBPL, such a language-sensitive tool (DBPL-USE[1]) can be pushed very far by exploiting advanced software generating and synthesizing tools [?, ?] based on attribute grammers [?].

Productivity in defining and extending computational objects benefits from the following properties of advanced editing environments:

- The amount of text that has to be entered by the user can be extremely reduced without producing cryptic code. The user merely selects one of the possible constructs at the current position instead of entering the text itself. The selection is carried out according to the definition of the language.
- The editor is working in a tree-oriented fashion. A top down approach in the development of design objects is therefore enforced.
- Layout rules and documentation standards may be set throughout a user community; the editor is completely responsible for the layout of the design objects (this is accomplished by *unparsing* rules).
- Most of the syntactic and many of the semantic details need not be remembered by the user. On selection of a menue item, a whole text pattern is inserted automatically.
- References to objects defined in related documents can be checked with respect to their appropriate use (it does not matter whether this feature is provided by the language definition or not).

[1] DBPL Usage Sensitive Environment, see also Fig. ??.

- Certain semantic errors and all syntax errors are discovered and reported at the earliest possible stage (on input).
- A wide range of semantic preserving transformations on the source text level can be done automatically.

The acceptance of the environment is improved by state-of-the-art interactive interface technology[2].

4.1 Compiler Generation Technology

Mainly due to two reasons, language editors qualify perfectly for the application of modern software generation technology. First, the knowledge about the target objects of a language-sensitive editor — programs in a specific language with its syntactic restrictions, type and scope rules etc. — can easily be formalized, e.g., by semantic actions or *Attribute Grammars*. Second, most of the language independent functionality of editors is "standard" and can be provided by a set of predefined library functions accessed by a *Synthesizer Generator*.

The two rival approaches to integrate semantic analysis into a language-sensitive tool can be characterized as follows:

Semantic actions: Every change of a design object implies the call of imperative routines that inspect and update the altered context. Every such routine must have a corresponding "inverse" one. The main problem concerning this approach is to determine the range of changes after the modification of a program. Advantageous is the use of global data structures (symbol tables), that can be read and updated being at an arbitrary node of the syntax tree. Therefore, the semantic action approach is very efficient concerning space. For compiler generating systems or, more generally, for all systems that work on complete source objects and do not need incremental updates, this approach is rather efficient.

Attribute Grammars: They describe properties of a language by means of *semantic equations*. Design objects are represented as attributed syntax trees. As opposed to the semantic action approach, the changes needed in case of a program modification are given imlipicitly in the formalism. Therefore, the design of attribute grammars does not have to consider the internal representation of a design object. Obtaining a new, consistently attributed tree after a modification is not a matter of design, but can be done automatically by means of an incremental attribution algorithm [?, ?, ?]. A major drawback of attribute grammars is their considerable space consumption. This is due to the fact that every node on the path between the node of information creation and the node of information use must store all the propagated information[3].

The *Synthesizer Generator* (henceforth called *sgen*), developed by Thomas Reps and Tim Teitelbaum at Cornell University, accepts an attributed grammar as its input and

[2] There exist (functionally equivalent) DBPL-USE editor versions for Sun View, Open Look and plain X-Windows environments.

[3] This problem seems to be overcome by means of *remote attribute updating*. The *synthesizer generator* permits a restricted form of such remote access.

constructs a *full screen editor* from this grammar. The editor accepts exactly the words (structured text objects) that can be derived by the given grammar and executes the semantic checks and actions specified there. The grammar must be written in a language that has been developed specifically for the definition of attribute grammars, namely the *Synthesizer Specification Language*. In order to process the specification and to construct the editor, the following actions are taken:

1. The grammar is parsed whereby completeness (exactly one semantic equation must exist for every nonterminal in the affected productions) and termination (every sequence of derivations must have a terminating production) are checked.

2. The generator tests the grammar for "orderedness". This can be done in polynomial time.

3. Lexical definitions (regular expressions) are passed to Lex [?] which constructs a finite automaton for the lexical analysis according to the specified regular expressions.

4. Parsing rules are translated into a form that is suitable as an input for the compiler generator Yacc [?]. Yacc is then called to construct a pushdown automaton (which is implemented as a procedure in *C* source code) according to these rules.

5. Evaluation strategies are computed for every possible situation and translated into *C* procedures.

6. Transformations and *unparsing* declarations are translated into appropriate sequences of function calls.

7. The created *C* program is compiled and the library functions supplied by the editor are linked.

The Synthesizer Generator Specification Language is a strongly typed, side-effect free language based on the concept of a *term algebra* defining sets of operator-operand trees (i.e. abstract syntax trees). The trees that can be derived from a given nonterminal are considered as *terms*. The operators of these terms denote nodes of a tree whereas the operands denote subtrees, hence terms again. Nullary operators are the atomic objects of these trees (leaves).

Since operators are associated uniquely with a production of the grammar, the definition of a set of trees may be considered as a type definition for the nonterminal which generates these trees. Attribute definitions are carried out in the same way and therefore attribute values are abstract syntax trees themselves. Simple predefined types[4] are (possibly infinite) sets of primitive trees (they only consist of the root of a tree).

For a more detailed discussion of the particular DBPL-USE specification and generation process see [?].

4.2 The Language-Sensitive Editor DBPL-USE

In a multi-lingual and semi-formal environment like DAIDA there will always be a need for the production of language texts under user control. On the CML level, this is the

[4] The predefined simple types are integer, real, bool, char, string, and references.

only way to get information into the system, but also on lower levels pieces of code will have to be produced manually. On the DBPL level, for example, there may be specific sets of formally justified *definition modules* for which – at a given system development stage – the *implementation modules* may be hand-coded or re-coded, e.g., for prototyping or for reasons of performance. DBPL-USE supports those needs for "programming in the small" by enforcing the interfaces of the definition modules plus all the syntactic and semantic rules of the DBPL language. Furthermore, it enforces the standards that have been set for program documentation and layout.

A text that has been edited by DBPL-USE is known to be more than just a sequence of characters; instead, the global knowledge base can be made aware of the fact that the text is a DBPL program which meets guaranteed semantic and syntactic criteria. The extent to which the knowledge represented by the tool may become explicit and common knowledge via the global knowledge base depends, of course, on the knowledge representation framework used in both components.

The central objectives of the implementation of DBPL-USE are semantic support for the strong typing of DBPL and syntactic support for expanding and restructuring of programs. Thereby, the user should have maximal freedom in choosing his style of editing. This is achieved by enabling menue driven insertions of every possible language construct as well as text input and parsing even for structures of low granularity. Furthermore, not only program expansions can be accomplished by selecting menue items but also restructurings of DBPL statements, simplifications of boolean expressions and derivations of (frames for) implementation modules out of definition modules and vice versa.

The DBPL-USE objectives are accomplished by extracting the context information from DBPL language constructs and using it to check contextual constraints which can be roughly characterized through the notions *type*, *scope* and *mode*:

Type Rules Since DBPL is a *statically typed* language, complete type consistency checks can be done at compile time. DBPL-USE utilizes the incremental availability of typing information to do type checks at the earliest possible point in time, i.e., at edit time. The uniformity of the DBPL type rules across all DBPL objects — from volatile computational data (e.g. boolean variables) to persistent databases greatly improves the exploitation of type information by the editor.

Scope Rules The *scope* of an identifier is the part of the program text in which it may be referenced. Scope rules determine the various scopes depending on the positions where the identifiers are declared. As it is the case in DBPL, the possible regions can also depend on the kind of usage (e.g., use within statements or within declarations). The application of scope rules associate every applied occurence of an identifier uniquely with the proper declaration. The scope rules of block-structured languages like Pascal or C are refined in DBPL as follows: (1) the scope of *exported* identifiers can be extended by import declarations to other, non-nested scopes; (2) identifiers introduced in *for each* and *with* statements or in quantified expressions (*some, all, each*) have a scope that is local to individual subexpressions or statements.

Sophisticated scope rules (like the rules encountered in DBPL) support higher levels of abstraction in the organization of large systems. The concept of locality allows programmers to concentrate on a small, understandable set of names, abstracting from declarations which are valid elsewhere in the system.

Object Modes The *mode* of an object denotes the possible usage of this object in the context of expressions, statements, declarations etc. In this way, additional control concerning accidental use is imposed. In programming languages the mode usually is declared by a keyword preceding the object declaration (e.g., **type**, **var**), but some modes are given implicitly. For instance, the declaration of a procedure defines a *constant* object. This object may be used as a statement or in assignments to procedure variables but not as the target of an assignment. DBPL provides the following object modes: constant, variable, type, value parameter, variable parameter, field, tagfield, procedure, transaction, selector, constructor and module.

4.3 A Session with DBPL-USE

The editors functionality is illustrated by a small but typical editor session. If the editor is started without any input file the following pattern is displayed in a newly created window:

module <*name*>
begin
  <*statement*>
end <*name*>.

The symbols included in "< >" are placeholders refering to nonterminals of the DBPL grammar. They denote positions where the program is syntactically incomplete.

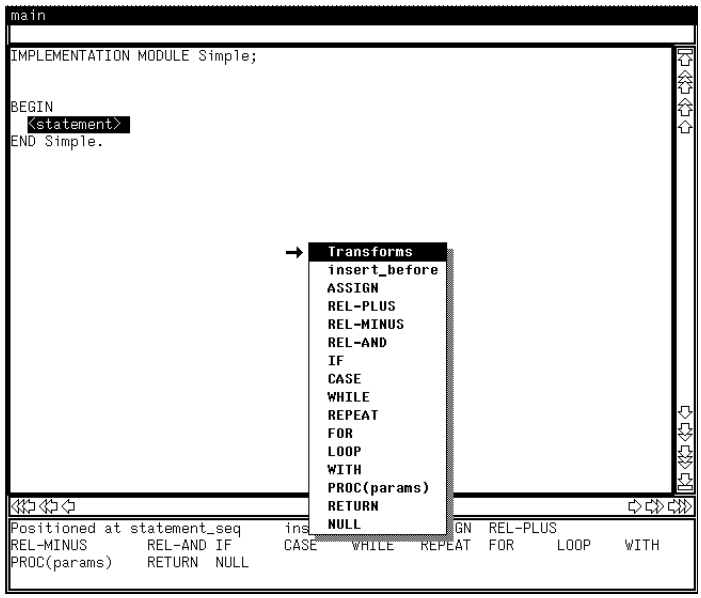

Figure 8: Pop-up-menue and transformation selection

From the transformations enabled at the current source text position – corresponding to the root of the DBPL grammar – the expansions *Def-Module* and *Imp-Module* and

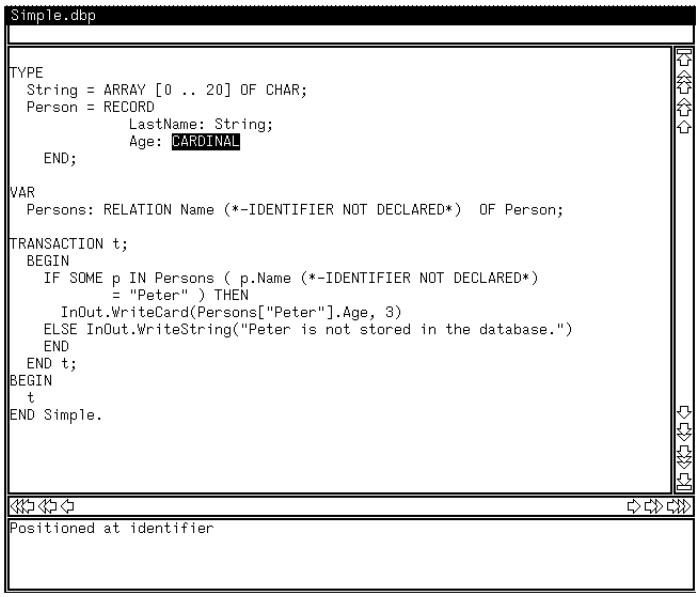

Figure 9: Change propagation and incompatibility messages

the option *Database* are enabled. The user selects *Imp-Module*. After inserting the module name and selecting the (not yet expanded) node <*statement*>, the user chooses to display the actual transformation menue to choose a DBPL statements to be inserted at this source code position (see Fig. ??). After a few more steps, the user has expanded the program by some type, variable and transaction declarations. By renaming the attribute *Name* in the record type *Person*, two program inconsistencies are detected by the editor and immediately tagged by the error message *"identifier not declared"* within the source text (see Fig. ??).

The current DBPL-USE implementation includes full recognition of the extended type-complete syntax of DBPL and provides full type and mode checking as well as control of internal import and export of DBPL objects.

Under the assumption that for some applications the user community of a specific *database* may exceed that of a certain (special-purpose) *language*, we are considering specializations of DBPL-USE to editors that are sensitive to specific sets of DBPL modules, thus providing editors that provide programming against specific DBPL databases.

5 On Next-Generation DBPLs

The DBPL system demonstrates how very expressive and orthogonal database programming languages can be implemented by a judicious combination of today's programming language and database technology, achieving a high degree of data integrity, program efficiency and user productivity.

Recent advances in programming language research towards expressive type systems, user-defined iteration abstractions and highly effective compilation schemes indicate that

it may be possible for next-generation DBPLs to support specific database programming requirements (declarative bulk data manipulation, transparent persistence management, concurrent transactional database updates) without resorting to massive built-in functionality like in the DBPL system.

In our current work in database programming language design and system implementation we therefore investigate *systematic* approaches to support typical database requirements (data independence, global query optimization) for user-defined, generalized bulk data structures (lists, trees, dictionaries, collections of two-dimensional objects) [?, ?]. A challenging research issue is to find adequate linguistic support to smoothly integrate database application and database system programming in order to assist the methodical construction of system extensions that are necessitated by database language extensions.

References