The Rationale behind DBPL *

Joachim W. Schmidt Florian Matthes

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

Abstract

The DBPL language orthogonally integrates sets and first-order predicates into a strongly and statically typed programming language, and the DBPL system supports the language with full database functionality including persistence, query optimization and transaction management. The application of modern language technology to database concepts results in new insights into the relationship between types and schemas, expressions/iterators and queries, selectors and views, or functions and transactions. Furthermore, it allows the exploitation of type theory and formal semantics of programming languages and thus connects database application development with results from program specification and verification.

1 On Data-Intensive Applications

The development of data-intensive applications, such as information systems or design applications in mechanical or software engineering, involves requirements modeling, design specification, and, last but not least, the implementation and maintenance of large database application programs. Although the database programming language DBPL concentrates on the last issue and offers a uniform and consistent framework for the *efficient implementation* of entire database applications, DBPL was designed, extended and evaluated with a specification and design methodology in mind.

Data-intensive applications may be 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 current data-intensive applications up to $O(10^9)$ or even higher. In strong contrast to the need for global management of large amounts of *representational data*, data-intensive applications also have

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

a strong demand 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 – and the need to cope with all these demands within a single conceptual framework that has to guide the design of a database programming language.

DBPL [SEM88, SM90a] is a successor of Pascal/R [Sch77] and addresses the above issues by incorporating a set- and rule-based view of relational database modelling into the well-understood system programming language Modula-2. DBPL extends Modula-2 into three dimensions:

- bulk data management through a data type *set (relation)*;
- abstraction from bulk iteration through *access expressions*;
- *persistent modules* and *transactions* for sharing, concurrency and recovery control.

2 Conceptual DBPL Foundations

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

Classical *type systems* and the *relational data model* constitute 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 exciting new experiences are made at the borderline where types (and expressions, selectors, functions ...) and data models (and queries, views, transactions ...) meet and have to be understood, one in the light of the other.

During DBPL development, several, mostly unnecessary and ad-hoc restrictions on both sides became obvious. Some limitations, as, for example, the lack of data *type orthogonality* (i.e. the restriction to first normal form relations) had already been realized, others, like *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). Other insights as, for example, the relationship between abstract access expressions and queries, iterators and views, or between recursive access expressions and deductive databases, came as surprises, at least to us.

Globally speaking, the DBPL language and system both heavily rely on conceptual and theoretical input from following areas:

Programming Language Foundations: typing, scoping, binding; [MS89, SM90b]

Compilation Technology: type checking, local data flow analysis, intermediate languages, separate compilation; [SM91]

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; [JK83]

Recursive Query Evaluation: fixed point semantics, stratification, recursive rule definition; [ERMS91]

Concurrency Control: multi-level concurrency control and recovery for complex objects; [SM91]

Distribution Models: transaction procedures, compensating transactions [JGL+88, JLRS88].

In summary, one can say that the DBPL project stayed more or less inside the boundaries drawn by conventional language technology (e.g. by *mathematical* typing schemes) and concentrated on improving the linguistic support for state-of-the-art database technology. Currently, however, we are heavily intrigued by novel programming language concepts (e.g. by *taxonomical* typing schemes [Car88]) and are quite positive that they provide the basis for substantial extensions of database technology [MS91].

3 Language Design Considerations

The DBPL language design is also influenced by requirements of the DAIDA environment for database application development [SWBM89, BJM+89, JMW+90] that asks for a target language that makes extensive use of sets and first-order predicates, and provides a complete separation of typing and persistence.

An essential guideline for the design of DBPL can be characterized by the slogan "power through orthogonality". Instead of designing a new language from scratch (with its own naming, binding, scoping and typing rules), DBPL extends an existing language and puts particular emphasis on the interoperability of the new "database" concepts with those already present in the programming language. In particular, DBPL aims to overcome the traditional competence and impedance mismatch between programming languages and database management systems by providing

- a uniform treatment of volatile and persistent data (there are, for example, relational variables local to DBPL procedures or as function parameters);
- a uniform treatment of large quantities of objects with a simple structure and small quantities of objects with a complex structure, as well as
- a uniform (static) compatibility check between the declaration and the utilization of each value.

Implementation details (e.g., storage layout of records, clustering of data, 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 runtime system choose appropriate implementation strategies based on "high-level" information extracted from the application programs. 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 also follows Modula-2 by occasionally sacrificing "language orthogonality" out of engineering and efficiency considerations. For example, DBPL does not support persistence of pointers and procedure variables. 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 [BJS86, JLRS88].

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 is further excelled by its balance between simplicity and expressive power. Finally, our group had a long tradition in Modula-2 compiler construction.

A design decision with far-reaching consequences is the compatibility between DBPL and Modula-2. DBPL is designed to be fully upward compatible with Modula-2, i.e. every correct Modula-2 program has to be correct DBPL program. This decision not only limits the freedom in language extensions (e.g. the keyword *SET* is already used for bit sets in Modula-2 and is not available for "true" sets in DBPL), but also forces adherence to language mechanisms (e.g. 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 to smoothly integrate DBPL into existing, fully-fledged software development environments (e.g. debuggers, profilers or version managers), to benefit from software libraries and from the interfaces to a variety of 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 environment (see DAIDA environment, e.g. [JMW+90]).

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 covering a wide range of operational database demands, such as query optimization, transaction and distribution management and computer-aided support for database application development. Readers interested in specific language and system aspects are referred to the following publications [JLS85, SEM88, JGL+88, MS89, SGLJ89, SM89, SM90a, SM90b, SM91]

4 The DBPL System and its Use

The DBPL project always had a strong commitment to implementability. A multi-user DBPL system under VAX/VMS is in use at the Universities of Frankfurt and Hamburg since 1985 for lab courses on database programming. There exist several DBPL system extensions that experiment with alternatives for concurrency (optimistic, pessimistic and mixed strategies) [BJS86] and integrity control [Böt90], storage structures for complex objects, recursive queries [JLS85, SL85] and distribution [JLRS88, JGL+88]. The construction of a distributed DBPL system is based on ISO/OSI communication standards and involves, for example, a re-implementation of the DBPL compiler to generate native code for IBM-PC/AT clients in cooperation with VAX/VMS servers.

During the last year a substantial effort was made to integrate the experience gained with these prototypes into a new, portable implementation of the DBPL runtime system on various platforms (VAX/VMS, Sun-3, Sun-4/Unix). By utilizing Sun's optimizing compiler backend, the DBPL compiler achieves "production-quality" performance and interoperability. Implementation aspects of the various layers of the DBPL database system are discussed in [SM91].

We expect the DBPL language and system to be used in research and devleopment mainly for the following three tasks:

Concept validation: As outlined above, it is often impossible to estimate the value of a proposed system solution (e.g. of a new concurrency control protocol or a new workstation-server architecture) by a single figure (e.g. maximum transaction throughput or number of packets to be transmitted). In many cases, the interaction with several system components (e.g. the recovery management or the query optimizer), or the lack of universality severly impairs the utility of a seemingly advantageous paper-and-pencil solution.

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

Application prototyping: From Modula-2 the DBPL language has inherited software engineering qualities that can not be found in commercial database environments and which qualify DBPL as an appropriate tool for designs and implementations. This use of DBPL is further supported by the quality of the commercial platforms (DEC/VAX/VMS and SUN/RISC/UNIX) on which the DBPL system is realized and the depth of its integration into professional environmets for software development and maintenance.[1].

References

[BJM+89] A. Borgida, M. Jarke, J. Mylopoulos, J.W. Schmidt, and Y. Vassiliou. The Software Development Environment as a Knowledge Base Management System. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, Topics in Information Systems. Springer-Verlag, 1989.

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

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

[Car88] L. Cardelli. Types for Data-Oriented Languages. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1988.

[1]The DBPL system is ready for external distribution through Hamburg University as of late spring 1991

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

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

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

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

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

[JMW+90] 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.

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

[MS91] 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 (to appear)*, Lecture Notes in Computer Science, 1991.

[Sch77] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), September 1977.

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

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

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

[SM89] J.W. Schmidt and F. Matthes. Advances in Database Programming: On Concepts, Languages and Methodologies. In *Proc. 16th SOFSEM'89*, Ždiar, High Tatra, ČSSR, December 1989. Available through Hamburg University.

[SM90a] J.W. Schmidt and F. Matthes. DBPL Language and System Manual. Esprit Project 892 MAP 2.3, Fachbereich Informatik, Universität Hamburg, West Germany, April 1990.

[SM90b] 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.

[SM91] J.W. Schmidt and F. Matthes. Naming Schemes and Name Space Management in the DBPL Persistent Storage System. In *Proc. of the Fourth Int. Workshop on Persistent Object Systems.* Morgan Kaufmann Publishers, January 1991.

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