Building Persistent Application Systems in Fully Integrated Data Environments: Modularization, Abstraction and Interoperability*

Joachim W. Schmidt Florian Matthes
Universität Hamburg, DBIS
Vogt-Kölln Straße 30
D-22527 Hamburg, Germany

Patrick Valduriez
INRIA
Roquencourt, B.P. 105
F-78153 Le Chesnay Cedex, France

Abstract

Research and development in the FIDE project on Fully Integrated Data Environments has led to the concept of Persistent Object Systems (POS) which generalize database systems by re-interpreting schemas as type definitions and databases as typed variables in addition to treating lifetime as a type-independent property. Furthermore, FIDE develops uniform linguistic interfaces for data modelling, computation and communication, and extends database, programming and communication technology to enable integrated application development based on interoperating POSs.

As a consequence of such an integrated view, formerly disjoint concepts such as databases, program and module libraries, files or repositories can now be treated uniformly as POSs differentiated essentially by the types of objects they contain and by the operational abstractions they provide.

Based on state-of-the-art database technology, this paper outlines FIDE results in extending databases and providing the integrated technology considered necessary for the construction and maintenance of Persistent Application Systems. Since our main goal is to improve substantially a system's capability of persisting successfully over time in changing environments, particular emphasis will be placed on systems scalability and its consequences for POS interoperability.

1 Introduction: Application Development in Fully Integrated Data Environments

Successful application development nowadays is rarely based on extensive *coding* of application programs. Instead, there is a strong tendency to application systems *modelling* by exploiting the services provided through open and modular environments already populated with prefabricated and packaged functionality and information. Examples of such environments are databases, function libraries and module repositories.

This shift in application development has motivated *service suppliers* to improve their market by providing safer and functionally extended servers, in addition to allowing *service consumers* to conveniently buy functionality and information by simply specifying their needs (— and paying for it). However, since such services are developed independently

*This research is supported by ESPRIT Basic Research, Project FIDE, #6309.

of each other with widely varying conventions on naming, typing, binding and whatever else may describe their interfaces, application developers who wish to exploit multiple services within a single application find themselves working in quite complex and finally unfriendly and unsafe environments. Examples are interfaces between C programs and SQL databases, window system or RPC packages.

In the ESPRIT Basic Research project FIDE our response to the situation sketched above is the development of a technology for Fully Integrated Data Environments. Central to the FIDE Research and Development is the concept of Persistent Object Systems (POS) which can be seen as a generalization of database systems along the following lines:

- re-interpreting database schemas as type definitions and databases as typed variables,
- treating lifetime as a property of computational objects which is independent of their type,
- developing uniform linguistic interfaces for data modelling, computation and communication,
- extending database, programming and communication technology to enable integrated application development based on interoperating POSs.

As a result of such a generalized view, formerly disjoint concepts such as databases, program and module libraries, files or repositories can now be treated uniformly as POSs differentiated essentially by the types of objects they contain and by the operational abstractions they provide.

Persistent Object Systems are required to provide the basis for characteristic interoperation needs such as:

- Persistent Object Management: Objects which outlive program executions or exist independently of any application impose specific requirements on their naming, scoping and binding mechanisms. Such mechanisms have to cope not only with the fact that object creation and object use may happen at different times but also on different platforms using diifferent tools.
- Data and Function Control: Within a single language, important classes of constraints on data and functions can be expressed by various type systems and for many of them there exist efficient algorithms to prove constraint satisfaction. Similarly, language-independent mechanisms have to be provided for persistent objects shared across platforms.
- Generic Interfaces and Functionality: The degree of re-usability of Persistent Object Systems is strongly correlated with the genericity of the services offered by the POS. Querying, report generation, form management, sorting and searching are examples of services accomplished best by instantiating generic algorithms with type-specific arguments. On the system level, substantial portions of interoperation protocols, e.g. link functions or marshalling procedures, can be obtained by providing generators with object signatures and platform specific information.

This presentation outlines FIDE results in providing the integrated technology considered necessary for the construction and maintenance of Persistent Application Systems as interoperating POSs. Section 2 reports briefly on the state-of-the-art of the integration and distribution technology developed in the context of Database Management Systems. The scope of the discussion is widened in section 3 by outlining a framework for interoperability across a variety of services represented by Persistent Object Systems. A specific environment for database programming with particular emphasis on interoperability across sites, languages and platforms is discussed in section 4. The paper concludes with a short reference on requirements for open communicating environments[1].

2 Overview: Distribution and Integration of Database Functionality

The success of Database Management Systems (DBMS) as commercial software products lies principally in their well-packaged functionality required by many bulk data processing applications: controlled persistent storage and optimized concurrent iteration. Although the concentration on a particular, well-chosen functionality is a major cornerstone of the DBMS success, it also contains the roots for its limitations. Marketed as closed, sealed servers, DBMSs contribute little to the interoperability *across* services. DBMSs interoperate with other systems only through their built-in interfaces which are either limited in functionality (e.g. SQL) or low in technology (e.g. cursor interfaces). Therefore, applications which require a variety of services, e.g. for objects on disk, on the screen, in memory or on wire, usually end up in low-level interface coding on the basis of strings, addresses and error codes.

On the other hand, in recognition of this shortcoming the database community is further improving interoperability *between* databases. This work can be classified roughly as covering two dimensions: the vertical dimension addresses interoperability issues between components of one logically related database transparently managed by a DBMS distributed over the sites of a network, while the horizontal dimension researches interoperability in the setting of multiple independent databases.

In this section we provide a short overview of the state-of-the-art of both dimensions in addition to reporting on research in the combined areas of distributed multidatabases. In the subsequent sections we open the discussion and outline a framework for interoperability across a variety of services, present a particular system for interoperable database programming and conclude with requirements for open communicating environments.

2.1 Databases as Distributed Systems

A distributed database (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network [ÖV91, ÖV93]. A distributed database management system (DDBMS) is then defined as the software system that manages a DDB transparently according to the following assumptions:

- Data are stored at a number of sites. Each site is assumed to consist logically of a single processor.
- The processors at individual sites are interconnected by a computer network thereby realizing a loose interconnection between independently operating processors.

[1] Earlier versions of the material presented in this paper can be found in [SM93, ÖV91, MS91]

- Data in a DDB are logically connected by relationships defined according to some structural formalism and accessed at a high level via some common interface (e.g. the relational model and its query languages).

- The system has the full DBMS functionality. It is neither a distributed file system nor a transaction processing system [Ber90].

These assumptions are valid for today's technology. Most of the existing distributed systems are built on top of local area networks with sites consisting of a single computer. However, next generation DDBMS environments will include multiprocessor database servers connected to high speed networks linking them and other data repositories to client machines that run application code as well as participating in the execution of database requests [Tay87, KL89, ZM89].

A distributed DBMS as defined above is only one way of providing database management support for a distributed computing environment. In [ÖV91] a working classification of possible design alternatives was presented distinguishing three dimensions: autonomy, distribution, and heterogeneity.

- Autonomy refers to the distribution of control indicating the degree to which individual DBMSs can operate independently. It involves factors such as information exchange between component systems, independent transaction execution, and the degree of individual DBMSs modification.

- Distribution deals with data. Usually two cases are considered: physical data distribution over multiple communicating sites or data concentration at only one site.

- Heterogeneity in distributed systems ranges from hardware heterogeneity and differences in networking protocols to variations in data managers. DBMS heterogeneity relates to data models, query languages, interfaces, and transaction management protocols.

Distribution is identified as one of the major characteristics of next-generation database systems, which will be ushered in by the penetration of database technology into new application areas with different requirements than traditional business data processing and the technological developments in computer architecture and networking. The nature of distribution in next generation DBMSs is a controversial issue. There are a number of documents that attempt to define alternative positions [SSU90, ABW+90, SRL+90, Sto90] all sharing at least the following two characteristics:

- Data model extension: Future data models need to be more powerful than the relational one, yet without compromising its advantages (data independence and high-level query languages). When applied to more complex applications such as CAD/CAM or software design the relational model exhibits severe limitations in terms of complex object support, type system and rule management. To address these issues, three important technologies, persistent polymorphic languages, object-oriented databases and knowledge base management are currently undergoing heavy research.

- Improved interoperability support: An expected consequence of applying database technology to an extended range of application domains is the proliferation of different, yet complementary, DBMSs and other generalized information services. Thus,

interoperability of such systems within a computer network becomes increasingly important. Fully integrated data environments require significant improvements in interoperability. In the following subsections, interoperability is addressed in more detail showing that it is receiving increasing attention in the joint community of databases and programming languages.

2.2 Integrated Views on Multiple Databases

As already indicated above, a *multidatabase* organization is an alternative to logically integrated distributed databases. The fundamental difference between the two is the degree of autonomy afforded to the component data managers at each site. While the independent DBMSs may have, for example, facilities to execute transactions, they have no notion of executing distributed transactions that span multiple components (i.e., they do not have global concurrency control mechanisms or distributed commit protocol implementation). The integration issue of multiple databases definitely deserves more space than available in this paper. In this subsection, we concentrate on issues of global database schema management and highlight some of the potential of object-oriented software for systems with high autonomy .

2.2.1 Global Schema Definition and Management Issues

The arguments against full transparency gain more weight in multidatabase environments. The additional autonomy of individual components and their potential heterogeneity make it more difficult (some claim impossible) to support full transparency. A major difficulty relates to the definition of a global conceptual schema (GCS) as a specification of the structure and constraints of the global database. The definition and role of the GCS is well understood in the case of integrated distributed database systems: it defines a logical single image over physically distributed data. The same clear understanding does not exist in the case of multi-DBMSs with autonomy [SY90]. One way the integration issue can be tackled is to treat the global conceptual schema as a generalization defined over local conceptual schemas. Studies along this line have been conducted before [MB81, DW84] and their practical implications with respect to transparency need to be reconsidered.

Finally, there are arguments that the absence of a GCS may be a significant advantage of multi-DBMSs over DDBMSs [LA87]. A language sufficiently powerful to access multiple databases without defining a GCS is presented in [LA87]. A significant research problem is the nature of such multi-DBMS languages. Problems with languages that are as powerful as the union of the component ones arise, as discussed in [SY90], with non-standard query operators of component DBMSs or with component DBMSs providing different interfaces to general-purpose programming languages. A solution may lie in extensible languages.

2.2.2 Object-Orientation and Multidatabase Interoperability

A central issue is the development of software technology that can deal inherently with autonomous system components. A prime candidate technology is object-orientation. Here we only comment on its role in addressing the interoperability issues. Object-oriented systems treat the entities in their domain typically as instances of some abstract type. In the case of database systems these entities are usually data objects. However, it is quite common for object-oriented models to treat every entity uniformly as objects. Thus, for example, interface definitions, user queries, or programs are considered objects in certain

object models and the technology is primarily concerned with providing the necessary tools and methodology for the consistent management of such entities.

Applying object-orientation to interoperability leads to the systematic and abstract treatment of autonomous DBMS components as software objects which can interoperate only through well-defined interfaces. There has been some initial work on the requirements of such interoperation interfaces [Man90, SY90, HZ90], an issue which clearly needs and deserves further investigation.

2.3 Database Interoperability: Multiple Databases in Distributed Environments

Finally, since multiple databases are expected to be available in distributed environments, the two problem spaces sketched above cannot be treated independently. This subsection outlines some of the issues raised when queries and transactions are run against multiple databases in distributed environments and concludes with a short reference to standards and their relevance for interacting autonomous systems.

2.3.1 Query Processing

The autonomy and potential heterogeneity of component systems create problems in query processing and optimization. A fundamental difficulty is global optimization when local cost functions are not known and local cost values cannot be communicated. Proposals have been made to concentrate on semantic optimization based only on qualitative information [SY90], however, semantic query processing is not fully understood either. There seems to be a great potential for hierarchical query optimizers which perform some global query optimization and leave it to each local system to perform further optimization on the localized subquery. Although this partitioning may not lead to optimal solutions, it will improve the tractability of the optimization problem. The emerging standards may also make it easier to define global cost models as well as share cost information.

2.3.2 Transaction Processing

For transaction processing in autonomous multi-DBMSs, the following consensus on a global transaction execution model seems to emerge: Each component DBMS has its own transaction processing services (i.e., transaction manager, scheduler, recovery manager) and is capable of accepting local transactions and running them to completion. In addition, the multi-DBMS layer has its own transaction processing components (global transaction manager, global scheduler) in charge of accepting global transactions accessing multiple databases and coordinating their execution.

Autonomy requires that global transaction management be performed on top of the existing local transaction execution functions. Heterogeneity has the additional implication that the transaction processing components involved may employ different concurrency control, commit, and recovery protocols and the coexistence of local and global transaction further complicates scheduling [BS88]. If serializability is used as the correctness criterion, it has to be ensured that the serialization order of global transactions at each site are the same. Some solutions use global serializability of transactions as their correctness criteria (see, e.g., [Geo91, BS88]) while others relax serializability (e.g., [DE89, Bar90]).

A further difficulty is the development of reliability and recovery protocols for multi-DBMSs and their integration with concurrency control mechanisms. Even though the

topic has been discussed in some recent works [BO91, Geo90, WV90], these approaches are initial engineering solutions.

2.3.3 Autonomy and Standards

Probably one of the fundamental impediments to further development of distributed multidatabase systems is the lack of understanding of the nature of autonomy. An initial characterization that has been made identifies three different forms of autonomy: design autonomy, communication autonomy, and execution autonomy. Other characterizations have also been proposed [GMK88]. Furthermore, most researchers treat autonomy as if it were an all-or-nothing criterion and not as a range of possibilities. It seems essential to precisely

- isolate a range of autonomy levels
- identify, on each level, the appropriate degrees of database consistency
- define transaction models appropriate for different levels of autonomy.

This might lead to a layered interoperability architecture for autonomous and possibly heterogeneous DBMSs, similar to ISO Open System Interconnection. Some work along this line is already under way within the Remote Data Access (RDA) standard, possibly improving substantially the development of practical solutions to DB interoperability.

3 Towards Fully Integrated Data Environments

Generalizing from database applications leads to the conclusion that application development benefits greatly from an integrated technology which supports modularized persistent systems and generalized module definition and management. In this section we present a higher-order core language which, essentially by modularizing name and binding spaces, supports module definition and interaction in the presence of module persistence. Reference is made to Modula-2 [Wir85], to DBPL [SM91, MRSS92a], a modular Database Programming Language with persistence, and to Tycoon [MS92, Mat93], which is used as a conceptual basis for our core language.

3.1 Modularity: The Basis for Interoperability

Based on the experience with Modula-2, the Database Programming Language DBPL exploits the power of modular programming and module management for data-intensive and long-lived applications. DBPL programs are structured into modules with well-defined import/export relationships. Definition modules provide signatures for type, value and location[2] bindings defined in implementation modules. Program modules export a single, parameterless function value, the main program.

An interface and a skeleton DBPL module exporting basic data types and values to handle telephone numbers would look as follows:

[2]Locations are the conceptual basis for sharing and updates. In this paper, however, we do not discuss the subtle type issues related to location bindings. The interested reader is referred to [SM93].

```
definition module Phone;
type Number = array [0..20] of char;
var localPrefix, myNr :Number;
procedure operator(prefix:Number):Number;
end Phone.
```

```
implementation module Phone;
procedure operator(prefix :Number) :Number;
begin ... return ... end operator;
begin (* module initialization: *)
   localPrefix:="4940";
   myNr:=concat(localPrefix,"85312");
end Phone.
```

During type checking of these compilation units, the DBPL compiler extracts type and value signatures from the definition module and verifies that the bindings defined in the implementation module match their corresponding signatures. In the next section we introduce a conceptual model that allows us to explain this process and the compilation of client modules against a compiled interface in more detail. This level of detail is required to generalize the simple Modula-2 module mechanism to distributed, multi-language database environments (see Sec. 4).

3.2 A Core Language for Fully Integrated Data Environments

In this section we introduce (informally) a model to describe the naming, typing and binding concepts involved in Persistent Object System interoperability. The model itself (subsequently referred to as the *Tycoon POS model*) is based on concepts of higher-order type systems and is sufficiently expressive to serve as a language-independent framework for program translation, generation and binding. The presentation of the actual DBPL system and its gateways in the following sections makes use of a limited subset of the Tycoon POS model. The potential of the full model is discussed in [MS93].

The Tycoon POS model is based on the notion of *types*, *signatures*, *values* and *bindings*. Types are understood as (partial) specifications of values. Values and types can be named in bindings for identification purposes and to introduce shared or recursive structures at the value and the type level. Signatures act as (partial) specifications of static and dynamic bindings. Bindings are embedded into the syntax of values, i.e. they can be named, passed as parameters, etc.

The syntax for *types* includes a set of base types B_i (**Int**, **Real**, **String**, ...), a type constant **Any** (the trivial type), user-defined type variables, function types, aggregated signatures, parameterized type expressions (type operator definitions) and type operator applications:

Type::= B_i | **Any** | *TypeName* | **Fun**(*Signatures*) *Type* | **Sig**(*Signatures*) |
Oper(*Signatures*) *Type* | *Type*(*Bindings*)

Function types are used to describe the signatures of parameterized objects like functions, procedures, methods, generators or relational queries. Aggregated signatures are used to describe the signatures of language entities like records, tuples, structures, modules, object definitions, database definitions, or object files.

Type operators denote parameterized type expressions that map types or type operators to types or type operators. Many programming languages have built-in type operators that map types to types (*Array*, *List*, *File*, *Pointer*, ...).

Signatures are sequences of value, location or type signatures. A value signature associates a value name with a type (*peter :Student*). A type signature associates a type name with a supertype specification (*Student <:Person*).

Signatures::= {*TypeSig* | *ValueSig* | *LocationSig*}
TypeSig::= *Name* <:*Type* *ValueSig*::= *Name* :*Type* *LocationSig*::= **var** *Name* :*Type*

Aggregated signatures are used to describe named *declarations* as they occur in function headings, module signatures, **external** declarations in C programs or in database schema definitions.

Signatures specify invariants on bindings and allow the verification of the correctness of (value or type) expressions depending on names without having access to the actual binding in which the name is defined. For example, based on the signature *age :Int*, the type-correctness of the expression *age + 1* can be verified without having access to the actual value bound to *age*. Signatures therefore play a central role in type-safe module systems.

The syntax for *values* includes base values $b_{ij} : B_i$, like 0, 3.4, "xyz", **true**, a canonical value of the type **Any**, function values including built-in functions like +, –, * of type **Fun**(*x* :**Int** *y:* **Int**) **Int** and user-defined functions and aggregated bindings:

Value::= b_{ij} | **any** | **fun**(*Signatures*)*Expr* | **bnd**(*Bindings*)

The syntax **bnd**(Bindings) defines that aggregated value and type bindings are first-class values, generalizing classical concepts like value, function and type aggregation in records, structures, abstract data types or modules. The syntax of *expressions* in functions is deliberately not specified here in order to be able to describe POS involving multiple languages.

Bindings are sequences of type, value and location bindings. A type binding *Age*≡**Int** defines a name for a type. A value binding *pi=3.1415* defines a name for a value.

Bindings::= {*TypeBnd* | *ValueBnd* | *LocBnd*}
TypeBnd::= *Name*≡*Type* *ValueBnd*::= *Name*=*Value* *LocBnd*::= *Name*=**var**(α_i) *Value*

Bindings model type, constant, variable and function declarations in programming languages and instances of database schemas in database systems.

As usual, types are intended to classify values, and signatures are to classify bindings. Moreover, it is possible to define *generic* functions (functions that take type bindings as arguments), value-dependent type operators, abstract data types (bindings that contain partially-specified type components and operations on that type). The formal type rules for this model (defining well-formed types, the types of values, the subtype relationship between types and the signatures of bindings) become therefore quite subtle.

In traditional systems, types and signatures are handled exclusively at compile-time, while values and bindings only appear at run-time. In persistent systems, the distinction between compile-time and run-time blurs, and it becomes possible, for example, to inspect value and type bindings at compile-time, giving rise to powerful reflective algorithms [SSS+92].

We now return to the DBPL example presented in section 3.1 and demonstrate how its modules are modelled in the Tycoon POS model:

CompileEnv ≡ **Sig**(*Phone* <:**Sig**(*Number* <:**array**...
var *localPrefix :Number* **var** *myNr :Number* *operator* :**Fun**(*prefix :Number*) *Number*)
linkPhone :**Fun**(*imports* :**Sig**()) *Phone*)

The definition module is represented as a type signature which associates the interface name *Phone* (understood as a type variable) with a supertype that is an aggregate of all

types and values signatures exported by the interface. The implementation module is represented as a single function, *linkPhone*, that takes an aggregated binding of all imported module values (in this case an empty binding) and returns a binding that conforms to the interface signature *Phone*.[3]

The signatures of *Phone* and *linkPhone* are both elements of a flat name space modelled by a signature bound to the type variable *CompileEnv*. In a typical DBPL system implementation, this name space is managed implicitly via search paths to look up compiled interfaces, called "symbol files".

The code generated for the implementation module *Phone* is stored as a named binding in a flat name space that collects all compiled modules. This name space is represented in the Tycoon POS model as a named value *linkEnv* whose type matches the signatures specified by *CompileEnv*:

linkEnv = **bnd**(*linkPhone*=**fun**(*imports* :**Sig**()) ...)

This name space is also utilized by the DBPL linker to compose all module initialization functions that make up an executable application program. The main module

module *Main* **import** *Phone;* ... **end** *Main.*

is represented by the following link function:

linkMain :**Fun**(*imports* :**Sig**(*phone* :*Phone*)) *Main*

It leads to the following module initialization sequence:

initEnv=**bnd**()
phoneE=*initEnv*∪{*phone*=*linkEnv.linkPhone*(*initEnv*)}
mainE=*phoneEnv*∪{*main*=*linkEnv.linkMain*(*phoneE*)}

3.3 The Potential of Persistent Objects

This section demonstrates how the concept of *orthogonal persistence* [AB87] as found in DBPL extends the potential for interoperability along two new dimensions: sharing over time and sharing between multiple users.

DBPL introduces the notion of a *persistent module* (database module) to define persistent location bindings in a strongly-typed programming environment. For example, the following local changes marked by underscores are sufficient to turn *localPrefix* and *myNr* into persistent variables (compare Sec. 3.1).

database definition module *Phone;*
  ... *(* see Sec. 3.1 *)*
end *Phone.*

implementation module *Phone;*
  ... *(* see Sec. 3.1 *)*
database definition begin
  localPrefix:="4940";
  myNr:=concat(localPrefix,"85312");
end *Phone.*

[3] As we will see later, we would lose some modelling precision if we were to treat the implementation module as a simple binding of type *Phone*.

The compilation of these modules defines a collection of signatures that is identical to the non-persistent environment *CompileEnv* defined in Sec. 3.2. The import semantics of persistent modules differ from volatile modules: the link-time execution of the module initialization code *linkPhone* of a persistent module does not return a new set of bindings to newly created process-local locations (copy semantics), but returns bindings to locations that are *shared* between all programs importing the persistent module (reference semantics). Therefore, side effects created by one application on persistent variables are visible to other applications importing the same persistent module:

module *Main1* **import** *Phone;*
begin *Phone.localPrefix:= "004940"* **end** *Main1.*

module *Main2;* **import** *Phone;*
begin *Phone.localPrefix:=concat(Phone.localPrefix,"-")* **end** *Main2.*

The sequential execution of *Main1* and *Main2* would lead to the following location bindings:

bnd(*localPrefix*=**var**(α_1)*"004940-" myNr*=**var**(α_2)*"494085312"*
operator=**fun**(*prefix :Number)...*)

The statement sequence marked by the keywords **database definition begin** is executed only once during the lifetime of the database module *Phone*, namely before it is imported for the first time into a DBPL application program.

The main advantages of persistent modules lie in overcoming the naming, typing and genericity problems associated with file-based solutions [SM93] without introducing additional linguistic complexity in the programming environment. Since the name space populated by database modules is not only persistent but also *shared* between several programs, it is necessary to ensure that destructive updates of persistent data structures performed by concurrent applications are performed *atomically* to handle system and program failures graciously. Therefore, DBPL supports user-defined (parameterized) *transactions* to handle the concurrency-control, recovery and integrity issues based on standard database transaction models.

4 DBPL: An Interoperable Database Programming Environment

A modular environment with persistence and sharing is particularly suited to support long-lived applications which need large collections of data. Since, over time, such applications have a strong tendency to extend functionality and to proliferate information, they also require scalability into open and distributed environments. In this section we outline FIDE results in functionality extension and in interoperability support. We present interoperability examples across database servers, programming languages and system platforms.

4.1 Service Extension: Bulk Types in DBPL

In the process of building a POS, it is often necessary to handle large, dynamic homogeneous collections of objects (e.g., class extents). Furthermore, it is necessary to represent

relationships between object collections and to perform efficient, set-oriented update and retrieval operations. Database systems have been designed to provide specific modelling and system support for these tasks. Expanding on our running example, let us assume that there is a need to register variable amounts of telephone entries composed of person names and telephone numbers. This kind of information is adequately described by the following relational database definition:

createdb *SQLPhoneDB* ...
create table *register (name* **char***(50), num* **char***(21))*

The language SQL provides simple, efficient, declarative read and write access to the information held in the database (**insert into table**, **delete from table**, **select** ... **from** ... **where** ...). However, it turns out to be surprisingly difficult to access SQL databases from application programs, e.g., to use the *Fax* service to send a message to every person named "Smith". We do not want to go into the detailed problems of SQL host language embedding, but there arise numerous difficulties due to the differences in naming, typing and binding between SQL and host languages such as C, Cobol or Ada (see [SM93] for details).

DBPL overcomes these difficulties by using the persistence and modularization concepts described in Sec. 3.1 in addition to extending the language by a generic bulk types operator **relation** and predefined polymorphic operations on values of type relation. Due to the orthogonality of the DBPL type system, it is possible to define a richer set of data structures than it is possible in the classical relational model, however, this flexibility is not required here [SM91]. The SQL database schema is represented by the following persistent DBPL module:

database definition module *PhoneDB;* **import** *Phone;*
type *Entry* = **record** *name :String num :Phone.Number* **end**;
type *Register* = **relation** *name* **of** *Entry;*
var *register :Register;*
end *PhoneDB.*

Names have been assigned to all types to be re-usable in application programs importing *PhoneDB*. It is worth noting that, of course, more than one database module can be defined in a DBPL application, thus realizing the concept of multi-databases discussed in Sec. 2.2. DBPL also provides a rich set of set-oriented update operators and an extended relational calculus including recursion based on fixed-point semantics to express bulk operations [ERMS91]:

register:+ Register{{*"John", "249"*}};
print(Register{**each** *n* **in** *register:n.num*>*"240"*});

DBPL has special (generic) type rules for the built-in relation operators, e.g., to capture the fact that the set-oriented insertion operator ":+" can be applied to relations of arbitrary element type, as long as the right-hand side expression is also a relation of the same element type. In [SM93] it is demonstrated how type operators of the Tycoon POS can be used to capture accurately these built-in type rules for relation types. This formalization is a only a first step towards a generalized type-safe handling of user-defined bulk data types [MS91, MS93].

4.2 The DBPL/SQL Gateway

The DBPL language and system supports bindings to external persistent objects in addition to internal persistent DBPL objects [MRSS92b]. For example, the following modification of the header of module *PhoneDB* binds the location variable *register* to an *external* SQL relation *register* defined in an Ingres database named *SQLPhoneDB*:

database definition for *Ingres* **module** *PhoneDB;*
import *Phone;* ... **var** *register :Register;* **end** *PhoneDB.*

All DBPL statements and expressions referring to external relation variables 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 (value and location bindings) as arguments and return (set) values that are converted appropriately for further processing within DBPL. For example, the query

if all *n* **in** *register n.phone* > *x* **then** ... **end**

is translated into a **select from where** SQL expression that uses the actual value stored in the DBPL location *x* of type **String**. 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.

Again, a conceptually simple generalization of an existing programming language concept suffices to overcome the interoperability deficiencies of todays database programming interfaces that have developed in a system-driven, bottom-up fashion.

Although the system details of the DBPL/SQL gateway are quite delicate and often require ad-hoc case analysis to achieve good system performance, this specific gateway implementation follows a more general pattern that directly reflects the model of typed programming languages in terms of types, signatures, values and bindings presented in Sec. 3.2. Our general experience in extending a language L_{int} by generic gateways to an external language L_{ext} (i.e., to embed L_{ext} as a sublanguage of L_{int}) is described in more detail in [SM93]. Specific requirements can be stated on the type and expression syntax of L_{int} and L_{ext} as well as on the tools for mapping signatures and bindings between L_{int} and L_{ext}. In the DBPL/SQL scenario this is achieved by using *DynamicSQL*, a set of library routines shipped with the Ingres DBMS to create cursors and communication buffers to convert Ingres values (element-by-element, attribute-by-attribute according to their type structure) to DBPL values and to pass arguments (of scalar types) to SQL query strings.

Our work on gateway construction can be summarized by the experience that the task is simplified considerably if the external functionality is provided through well-structured libraries with abstract and "minimal" signatures, and not through extensive, "verbose" (SQL-like) and informally described interfaces.

4.3 Cross-Language Interoperability

The most primitive (but also most common) form of cross-language interoperability is achieved by having a standardized, language-independent link format (e.g., COFF of Unix

System V) that allows static bindings in a language L_{imp} to bind to values or locations defined in another language L_{exp}. In this setting, L_{imp} is able to import from L_{exp}. The next step is to define standardized, language-independent parameters passing conventions that allow argument values or argument locations defined in L_{imp} to be bound dynamically to function parameters defined in L_{exp}. If the roles of L_{imp} and L_{exp} can be interchanged, full cross-language interoperability (including "call-back" mechanisms) is supported.

Since this interoperability takes place at the value, location and binding level only, all naming and typing consistency control enforced by the use of type names, types and signatures in compilers is effectively lost in this scenario.

The DBPL compilers for VAX, Sparc and Motorola architectures attack this problem by providing the DBPL programmer with a mechanism to recover type and signature information for external bindings via so-called *foreign definition modules*. For example, the interface of the *Fax* module defined in Section 4.4 could be revised as follows to define a FAX service implemented in the programming language C:

definition for *C* **module** *Fax;*
import *Phone;*
type *Status = (error, busy, done);* ...
end *Fax.*

The compiler will enforce the consistent use of the bindings exported by the external *Fax* package in all importing DBPL programs. For example, it would catch the following type error in the application of the function *Fax.dial* that attempts to pass an integer value as a string argument:

module *SendFax;* **import** *Phone, Fax;*
begin *Fax.dial(853228);Fax.send("Sample fax.");Fax.hangup();* **end** *SendFax;*

The skeleton of a C-program to provide value bindings matching the signatures *Fax* looks as follows.

typedef char* *Phone_Number*
typedef int *Status*
#define *error ((Status) 0)*
#define *busy ((Status) 1)* ...
*Status Fax_dial(***char*** number)*{ ... }
*Status Fax_send(***char*** text)* { ... **return** *done;* ... }
void *Fax_hangup()* { ... }

Since relation types and relation operations are fully integrated into the DBPL language, they can be freely combined with the cross-language binding mechanisms:

for each *b* **in** *register :contains(n.name, "Smith")* **do** *Fax.dial(n.num);* **end**

It is also possible to call DBPL transactions on bulk objects from C.

4.4 Cross-Platform Interoperability

The distributed version of DBPL [JGL+88, JLRS88] exploits the basic module concepts to add an additional layer of type-safety to standard remote procedure call mechanisms (RPC) in federated client-server programming models.

To give access to a local fax service at a site called "CentralOffice", this site would compile the following *remote definition module* and then *export* the compiled description (typically together with its source text for documentation purposes) only to those clients on the network who are to be authorized to use the service.

remote definition module *Fax* **for** *"CentralOffice";*
import *Phone;*
type *Status = (error, busy, done);*
procedure *dial(number :Phone.Number) :Status;*
procedure *send(text :***array of char***) :Status;*
procedure *hangup();*
end *Fax.*

In this scenario, signatures of compiled definition modules accumulated in the compilation environment serve as protocol specifications. Frequently clients require distribution transparency. In this case it is of considerable advantage if a main program

module *SendFax;* **import** *Phone, Fax;*
begin *Fax.dial("853228");Fax.send("Sample fax.");Fax.hangup();* **end** *SendFax;*

stays textually unchanged whether it uses a local definition module *Fax* or the above remote definition module *Fax* for "Central Office" offered by a server somewhere in the network. This distribution transparency is achieved as usual by a client stub and a server stub that marshal and unmarshal the arguments and results supplied to functions defined in the remote definition module.

In terms of the Tycoon POS model, RPC-based communication mechanisms are an implementation technology that enables the creation of function value bindings between names in a client program and function values in a server program. Using plain RPC mechanisms it is not possible to directly define location bindings spanning machine boundaries. In particular, we would have to revise the module interface *Phone* not to directly export the locations *localPrefix* and *myNr*. However, using the concept of persistent variables introduced in section 3.3 these restrictions are lifted in DBPL by the provision of truly distributed persistent variables.

The crucial feature of DBPL is to retain (static) type safety across machine boundaries by maintaining a distributed compilation environment that allows local and remote modules to *share* signatures for type checking purposes and to share module bindings for transparent connection establishment.

5 Concluding Remarks

After seven years of development, the DBPL system has now reached a level of maturity and interoperability that makes its linguistic abstractions readily available for implementors of non-trivial Persistent Object Systems on several hardware-platforms.[4]

From a research point of view, an interesting side-effect of this DBPL implementation effort is an insight into repeating patterns of language and system extension requirements, some of which are outlined in Sec. 4. Consequently, our current work in the Tycoon project [Mat93, MS91, MS92] investigates languages and architectures that facilitate such incremental, problem-specific extensions in a type-safe environment.

[4]The DBPL system is distributed by Hamburg University.

In contrast to DBPL, Tycoon takes a rather radical approach by not maintaining upward compatibility with existing programming languages (Modula-2) and data models (extended relational models). Also its internal protocols for store access, program representation and linkage do not adhere to pre-existing standards. The rationale behind the design of Tycoon is to provide a lean language and system environment that provides just the kernel services and abstractions needed to define higher-level, problem-oriented "languages" and "data models" and provide an "ideal" basis for systems extensibility and interoperability.

References

[AB87] M.P. Atkinson and P. Bunemann. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2), June 1987.

[ABW+90] M. Atkinson, F. Bançilhon, D. De Witt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Deductive and Object-oriented Databases*. Elsevier Science Publishers, Amsterdam, Netherlands, 1990.

[Bar90] K. Barker. *Transaction Management on Multidatabase Systems*. PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1990. available as Technical Report TR90-23.

[Ber90] P.A. Bernstein. Transaction Processing Monitors. *Communications of the ACM*, 3(11):75–86, 1990.

[BO91] K. Barker and M.T. Özsu. Reliable Transaction Execution in Multidatabase Systems. In *Proc. 1st Int. Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, 1991.

[BS88] Y. Breibart and A. Silberschatz. Multidatabase Update Issues. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, Illinois*, June 1988.

[DE89] W. Du and A. Elmagarmid. Quasi-Serializability: A Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, August 1989.

[DW84] U. Dayal and H.Y. Wang. View Definition and Generalization for Database System Integration in a Multidatabase System. *IEEE Transactions on Software Engineering*, SE-10(6):628–645, November 1984.

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

[Geo90] D. Georgakopoulos. *Transaction Management on Multidatabase Systems*. PhD thesis, Department of Computer Science, University of Houston, TX, 1990.

[Geo91] D. Georgakopoulos. Multidatabase Recoverability and Recovery. In *Proc. 1st Int. Workshop on Interoperability in Multidatabase Systems*, Kyoto, Japan, 1991.

[GMK88] H. Garcia-Molina and B. Kogan. Node Autonomy in Distributed System. In *Proc. Int. Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, December 1988.

[HZ90] S. Heiler and S. Zdonik. Object Views: Extending the Vision. In *Proceedings of the IEEE Sixth International Conference on Data Engineering*, 1990.

[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.

[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.

[KL89] W. Kim and F.H. Lochowsky. *Object-Oriented Concepts, Databases and Applications.* ACM Press Books, 1989.

[LA87] W. Litwin and A. Abdellatif. An Overview of the Multidatabase Manipulation Language MDL. *Proc. IEEE*, 75(5):621–631, May 1987.

[Man90] F. Manola. Object-Oriented Knowledge Bases – Parts I and II. *AI Exepert*, pages 26–36, 46–57, March and April 1990.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German).

[MB81] A. Motro and P. Buneman. Constructing Superviews. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 56–64, 1981.

[MRSS92a] F. Matthes, A. Rudloff, J.W. Schmidt, and K. Subieta. The Database Programming Language DBPL: User and System Manual. FIDE Technical Report FIDE/92/47, Fachbereich Informatik, Universität Hamburg, Germany, July 1992.

[MRSS92b] F. Matthes, A. Rudloff, J.W. Schmidt, and K. Subieta. A Gateweay from DBPL to Ingres. FIDE Technical Report Series FIDE/92/54, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[MS91] 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. (Also appeared as TR FIDE/91/27).

[MS92] 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.

[MS93] F. Matthes and J.W. Schmidt. System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. In P.P. Spies, editor, *Proceedings Euro-Arch'93*, 1993.

[ÖV91] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[ÖV93] M.T. Özsu and P. Valduriez. Distributed Data Management: unsolved problems and new issuses. In T. Casavant and M. Singhal, editors, *Readings in Distributed Computing.* IEEE Computer Society Press, 1993.

[SM91] J.W. Schmidt and F. Matthes. Modular and Rule-Based Database Programming in DBPL. FIDE Technical Report Series FIDE/91/15, Fachbereich Informatik, Universität Hamburg, Germany, February 1991.

[SM93] 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.

[SRL+90] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, and P. Bernstein. Third-Generation Data Base System Manifesto. *ACM SIGMOD Record*, 19(3):31–44, September 1990.

[SSS+92] D. Stemple, R.B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson, and S. Alagic. Type-Safe Linguistic Reflection: A Generator Technology. Research Report CS/92/6, Univ. of St. Andrews, Dept. of Comp. Science, July 1992.

[SSU90] A. Silberschatz, M. Stonebraker, and J.D. Ullman. Database Systems: Achievements and Opportunities, Report of the NSF Invitational Workshop on the Future of Database Systems Research. Technical Report TR-90-22, Department of Computer Science, The University of Texas at Austin, TX, 1990.

[Sto90] M. Stonebraker. Architecture of Future Data Base Systems. *IEEE Quarterly Bulletin Database Engineering*, 13(4):18–23, December 1990.

[SY90] P. Scheuermann and C. Yu. Report of the Workshop on Heterogeneous Database Systems. *IEEE Quarterly Bulletin Database Engineering*, 13(4):3–11, December 1990.

[Tay87] R.W. Taylor. Data Server Architectures: Experiences and Lessons. In *Proc. CIPS (Canadian Information Processing Society) Edmonton '87 Conf.*, pages 334–342, 1987.

[Wir85] N. Wirth. Report on the Programming Language Modula-2. In *Programming in Modula-2*. Springer-Verlag, 3rd edition, 1985.

[WV90] A. Wolski and J. Veijalainen. 2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase. In *Proc. Int. Conf. on Databases, Parallel Architectures and their Applications*, pages 321–330, Miami Beach, FL, 1990.

[ZM89] S.B. Zdonik and D. Maier. *Readings in Object Oriented Database Management Systems*. Morgan Kaufmann Publishers, 1989.