Data Persistence in Programming Languages
A Survey

Stewart M. Clamen
May 30, 1991
CMU-CS-91-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Database systems are primarily concerned with the creation and maintenance of large, long-lived collections of data, while traditional programming languages have promoted such ideas as procedural control and data and functional abstraction.
While each provides considerable utility in their respective domains, there exists a large number of applications that require functionality from both database and programming language systems. To this end, there has been serious effort over the past few years at developing systems that integrate the basic ideas from the two domains. This paper concentrates on research developments which have resulted in programming languages incorporating database functionality into their programming models, most importantly, a concept of **data persistence**.

This research was sponsored in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597; and by the Office of Naval Research under Contract N00014-88-K-0641. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Programming languages, database systems, data persistence

1 Background

Database systems are primarily concerned with the creation and maintenance of large, long-lived collections of data. More specifically, modern database systems are characterized by their support of the following features:

Data Persistence and Reliability: The ability for data to outlive the execution of a program, and possibly the lifetime of the program itself. Coupled with this feature is the assurance that the data in the database is protected from hardware and software failures.

Data Sharing: The ability for multiple applications (or instances of the same one) to access common data, possibly at the same time.

Scale of operation: The ability to operate on large amounts of data in simple ways.

Traditional programming language systems, on the other hand, provide facilities for procedural control and data and functional abstraction, but lack built-in support for any of the above database features.

While each provides considerable utility in their respective domains, there exist a large number of applications that require functionality from both database and programming language systems. Such applications, often called **design tasks**, are characterized by their need to store and retrieve large amounts of shared, structured data. Examples of such applications include CAD/CAM systems, office automation facilities, and software engineering systems. To this end, there has been serious effort over the past few years at developing systems that integrate the basic ideas from both domains. This paper concentrates on research developments that have resulted in programming languages incorporating database functionality into their programming models, most importantly, the concept of **data persistence**.

A **persistent programming language** is a language that provides to its clientele the ability to preserve data across successive executions of a program, and even allows such data to be used by many different programs. Data in a persistent programming language is independent of any program, able to exist beyond the execution and lifetime of the code that created it.

A **database programming language** is a language that integrates some ideas from the database programming model with traditional programming language features. Such a language is distinguished from a persistent programming language in that it incorporates features beyond persistence.

A primer on the features and nomenclature of relational database systems is included in an appendix to this paper.

1.1 Why Persistent Programming Languages?

Researchers working on the development of persistent programming languages have been motivated primarily by the following ideas:

1. The languages provided within existing database systems are not expressive enough, and a persistent programming language is the first step towards a fully integrated database programming language.

2. The lack of integration makes the writing of applications that depend on persistent data access difficult or impossible.

3. The presence of persistence is a useful addition to a programming environment.

The rest of this subsection elaborates on these points.

1.1.1 Towards a Database Programming Language

A well-known shortcoming of relational database systems is the lack of ability to enforce some important, general and application-specific qualities at the database level. In an effort to remedy this problem, Date has proposed an **integrity language**, which would provide a way of enforcing various constraints on record values. Some projects within the database research community have attempted to integrate some of his ideas into existing systems. More practical solutions use programs to detect or prevent the creation of illegal database states. [Dat81]

Limitations on the expressive power in traditional database languages is another consideration. The query and data definition languages associated with database systems are usually quite limited in their functionality. One major shortcoming is the lack of ability to perform transitive searches over the database. [AU79]

Many researchers recognize that an integrated programming language and database system could solve both of these serious database system deficiencies.

1.1.2 Some applications need database-like functionality

Computer applications, such as CAD/CAM systems, multimedia and graphics systems, and office automation, would benefit from having access to database features, such as a persistent store. Non-integrated approaches to accessing database functionality from within programming languages (*e.g.,* using standard operating system file systems as a medium for storage of long-lived data) suffer from poor performance.[Mos89] Also, the improved functionality and semantics that would come with an integrated system would be a positive improvement.

Atkinson[**ABC$^+$83**] [1] claims that a significant amount of programming effort (and code

[1] Reference citations are printed in boldface if they were used as primary sources for the featured persistent languages and systems. The reader can locate annotated bibliographic references to these papers using the index found at the end of this survey.

space) is devoted to converting data from database or file formats into and out of program-internal formats. The integration of a persistent store into the language would free the programmer from that responsibility.

1.1.3 Database functionality in a programming environment is useful

The addition of a persistent storage facility into a programming language system is a useful enhancement to an interactive development environment. Some existing interactive languages (LISP, ML[Har90], Smalltalk[GR83]) allow the user to checkpoint the current state of the heap for future use. A fully persistent store would free the user from the *all-or-nothing* aspect of this procedure, enabling him to store away specialized objects like program libraries or constructs of previously-created data. [**AM85**]

Programming environments in general are a class of application that would greatly benefit from some sort of persistent storage facility. A number of projects already feature research in this direction. [Rei86, HZ87, RWW89, HSS89]

1.2 A Brief History of Database/Programming Language Interaction

This section discusses two early efforts at data and programming language interaction, and why they are not sufficient for many classes of applications.

1.2.1 Embedded Database Languages

Many relational database systems provide the ability to textually insert database operations within traditional programming languages, usually COBOL or PL/I. The INGRES system supports a number of database languages, including SQL and QUEL, and provides the ability to embed database language commands within COBOL, FORTRAN, BASIC, Pascal, and C programs. Communication is through a set of variables in the host language, and forms to commit or abort a transaction[2] are provided. [Dat81, Dat87]

Although it does provides some useful functionality, the embedded language support is rather clumsy. To make effective use of the system, a programmer must be familiar with the eccentricities of both languages, as well as the eccentricities of the interface itself. Also, the programmer is restrained by the database language, being restricted to the primitive database data types for persistent storage. Programmers wishing to build applications that require database functionality on more advanced data types are out of luck.

1.2.2 Pascal/R: An Early Database Programming Language

The first attempt to provide some level of database functionality from within a programming language was made by Pascal/R.[Sch77] Unlike the embedded database languages, Pascal/R extends the syntax and semantics of the base language to support database operations. Interaction with the store is through a pair of new datatypes, `relation` and `database`.

[2] A brief description of transaction systems in included at the end of Appendix B.

`Databases` are similar to files: they must be explicitly opened and closed. Unlike the regular typed files of Pascal, which can be declared to contain instances of any Pascal type, `databases` are restricted to act as the (exclusive) repositories for `relation` instances, the other new Pascal/R data type. Similar to their namesakes found in relational database systems; `relation`s are tuples whose fields are restricted to atomic base types like `int` and `char`, and on whom primary indexes can be defined. An iteration construct, `each`, is provided for operations mapped over entire relations, such as queries or aggregation operations. `Database`s maintain the sets of (persistent) `relation` types.

Although well integrated into the base language, the database-inspired data types and control structures of Pascal/R are not universally applied. Instances of the other, more sophisticated data structures of Pascal are excluded from the persistent store. A programmer would be forced to convert his volatile structures into simple `relation`s, much the same way programmers of traditional programming languages must convert data into character strings before storing them into standard (operating system) files. Also, the aggregate semantics defined for relations is not supported for more standard Pascal types like set and sequence. (See Section 2.1.5 (p.10) for more on this idea.)

Pascal/R has other problems. The persistent store is implemented on top of a collection of files, and programs can only access one database file at a time, making sharing difficult. The transaction system is simplistic as well; entire programs run as single transactions. In spite of its shortcomings, however, Pascal/R was an admirable first step in this direction of research.

1.3 Efforts since Pascal/R

Inspired by both the features and shortcomings of Pascal/R, early integration work focused on expanding the pervasion of persistence in the programming language. Later efforts moved on to the building of more powerful persistent object systems, with the intention of providing support for such advanced database language requirements as collections and query processing. At the same time, researchers in the object-oriented community, recognizing similarities between the database and object-oriented data models, began the development of object-oriented database systems to support robust and efficient applications. A number of the more representative efforts in each of these areas will be explored in detail in later sections of this paper.

2 Focus

In this section, I introduce and discuss the language system features that represent the focus of my survey — presented in the form of questions. In the summaries of the subsequent sections, I provide answers to these questions for a number of representative persistent and database programming languages.

2.1 Persistence

Persistence is one of the primary motivations behind database systems and is an essential feature of any database programming language. It is therefore important to ask the following questions for any language I review:

- What is the view of the persistent store?
- What data types admit persistence?
- How does one declare/denote persistence?
- How does one locate (existing) persistent objects in the database?
- How does one manipulate persistent objects once they have been located?
- Is there support for the evolution of type definitions?
- How is persistence implemented by the underlying system?

The issues behind these questions are not independent; the model of persistent storage chosen by a language affects how they can be manipulated, for example. However, the answers to these questions do serve as a useful method for describing a language's concept of persistent value and binding.

2.1.1 What is the view of the persistent store?

Before an application can operate on a persistent store, some initialization procedure must be called, *i.e.,* the connection between program and store must be established. The various languages and systems can be categorized by which of the following three forms this initialization takes:

1. **Single Database, implicit database opening.** The most simple and transparent model, this scheme has the runtime system automatically establish a connection to the (single) store at startup, freeing the user from any initialization responsibility.
2. **Multiple Databases, explicit database opening.** The persistent store is partitioned across multiple, independent "files". This scheme forces the programmer to assume some of the responsibility of database management, requiring the explicit choice of partition(s).

3. **Multiple Databases, implicit database opening of a default database.** A special case of the previous scheme, this plan defines a default partition of the store, so that programs are free to ignore the issue of initialization altogether if they wish.

There is no semantic advantage to the design of explicitly partitioning the persistent store. In systems that support this, the partitioning is mostly provided as a means of cheaply supporting coarse-grain parallelism or fault tolerance.

Once the connection to the persistent store has been established, communication between the store and the application can be achieved in a variety of ways. Cardelli and MacQueen [CM88] describe three possible models for the store. They are, in order of increasing system complexity:

1. **Explicit import/export.** Specific operations are provided in the host language to transport (or copy) the object from the persistent to the volatile (working) area.
2. **Fully Transparent Access.** Persistent objects are imported from the persistent store on demand, much like pages in a virtual memory systems. Modified persistent objects are written back to store as appropriate. Although the transparency of this approach is attractive, it fails to address the issue of concurrent access.
3. **Mostly Transparent Access.** Total transparency of access is compromised here in favor of a mixed approach, which provides explicit operations to effectively handle the potential problems resulting from concurrent access to the store.

Amber [**Car86a**] is representative of the first model, while Galileo [**ACO85**] favors the fully transparent approach. All of the other languages surveyed here provide automatic interning of persistent objects with some explicit control over locking and concurrency.

The final issue characterizing the sophistication of the communication model concerns the matter of persistent object identity: Is there a one-to-one mapping between an object in the persistent store and its "working copy" in volatile memory when it is manipulated at runtime? In other words, is the persistent store preserving persistent *objects* or just persistent *values*?

Of the systems reviewed, only Amber's primitive storage model lacks identity-preserving semantics, but the issue is worth mentioning nonetheless. In relational database systems, the simple structures of the relations make the issue moot, as relations only admit immediate values.

2.1.2 What data types admit persistence?

In some persistent languages, persistence is a quality attributable to only a subset of the admissible language data types. For example, Pascal/R [Sch77], inspired by relational database systems, supports persistence exclusively to instances of type `relation`, a special record type whose elements are restricted to base types like numbers and strings (p.5).

PS-Algol [**ABC+83**] promotes the idea that the quality of persistence should be extended to all language data types. This approach, termed **orthogonal persistence** [Coc83], has been adopted by a number of systems, including Galileo and OPAL [**BMO+89**].

Some languages view persistence as a property of a data type and move to restrict persistence to a number of built-in and user-defined data types. This approach is exemplified by Amber, E [**RC89b**], and Avalon/C++ [**DHW88**].

2.1.3 How does one declare/denote persistence?

How does the programmer specify that a certain object should persist? Most language designers have attempted to integrate the declaration of persistence with their language's type system. Popular approaches include:

- When orthogonal persistence is used as part of the language model, persistence is not a quality attributable to a data type, but rather, to instances. As a result, the language must provide some runtime method for indicating which objects should survive. PS-Algol, with its tagged architecture and garbage collector, ensures that any object reachable (via pointer traversal) from a special root object will persist. This approach can only work for systems possessing some sort of tagging on data values.
- In E, a statically typed language based on C++, persistent types are declared using a special `dbclass` declaration, and instances are created similarly to volatile ones. Avalon/C++ and AVANCE [BB88] are other object-oriented languages which provide a special class hierarchy for persistent values.

In some languages systems, like Galileo, any data that is accessible at program termination persists, so there is no distinction between volatile and persistent data.

Let us compare the costs and benefits of orthogonal persistence against those of a persistence-by-typing scheme, as exemplified by languages such as E and Avalon/C++. In general terms, orthogonal persistence is more elegant, while the typing scheme is more efficient.

The uniform treatment of objects in a system based on the principle of orthogonal persistence is more convenient for the both the programmer and the system. Just as in traditional garbage-collected languages, where objects persist as long as they are addressable (or until the program terminates), objects in a language with orthogonal persistence survive as long as they are addressable from the persistent root. Galileo takes this idea to an extreme, defining the top-level environment as the persistent root.

However, there is some runtime expense in a system where every pointer reference might be addressing a persistent object. The system is required to test if the object must be loaded in from the disk-resident database. Also, orthogonal persistence promotes transparency, and, as mentioned on p.8, a system with support for sharing among concurrent processes cannot be fully transparent.

Having persistent type instances results in an interesting side-effect. In many strongly-typed languages, type information is required in the compilation phase, but is superfluous at runtime, and thus is discarded. When such a language is extended to support persistence, however, the type description must be made persistent along with its instances, so that type dependencies can be maintained across sessions. [CM88]

2.1.4 How does one locate persistent objects? How are persistent objects addressed?

With the large number of objects that any serious persistent language system must be able to support, an efficient and simple naming, or addressing, mechanism is a necessity.

Early approaches to this problem were in line with the way naming is supported in traditional programming languages and systems. PS-Algol and OPAL proposed forms of hierarchical directory structures, indexed by character strings, similar to file systems or Smalltalk environments. [AM84, MS87] Some projects, including E, attempted to integrate naming as part of the language's module system, binding persistent objects in a way similar to how dynamic loaders load procedures out of program libraries. [**RC89b**]

The chief strength of relational database systems is their ability to cope with large collections of objects uniformly. In such systems, primary keys are the basic addressing mechanism, allowing easy access to arbitrary table elements. Recent database programming languages attempt to solve the addressing problem along this line. Such languages typically support unordered, typed collections of (anonymous) instances. Coupled with mechanisms for efficient, query-like searches, an efficient and powerful addressing scheme can be designed. As an example, restricting elements of an unordered collection of class instances to be unique in the values of some of their properties results in an effective naming mechanism for each element in that collection.

2.1.5 How does one manipulate persistent objects once they have been located?

The first programming languages to interface with database systems did so by supporting the embedding of database query language constructs within a program. Communication of data values between the database language fragment and the programming language is very cumbersome and restrictive. One hopes that a more closely integrated persistent programming language would reduce, or totally eliminate this problem, termed **impedance mismatch** [MS87].

To minimize the mismatch, many persistent programming languages provide the ability to manipulate persistent data in the same manner as volatile data. In a language where persistence is applied at the object level, this would mean allowing persistent instances to be operated upon using the same functions and constructs as the corresponding volatile data. In a language with persistence as a property of a type, this would mean allowing users to define routines in the same manner as for a volatile type.

Other languages [**Car86a**, CM88] promote a dual dataspace approach, requiring the user to explicitly translate the object from persistent into active memory before it can be acted upon. Likewise, the user wishing to install an object (new or modified) into the store must explicitly write it back.

2.1.6 Is there support for the evolution of type definitions? How are instances of outdated type descriptions handled?

In traditional programming systems, abstract data types are often used to reduce the amount of dependency between the data type programmer and the data type client. If the type designer chooses to reimplement a type, clients will not be required to reprogram, so long as the abstract interface is preserved. In a language with persistence, however, data type clients may have already installed instances of the outmoded type in the persistent store. Discarding the old type and its instances is hardly prudent, and rather than force the programmer and user to contend with old instances indefinitely, a database programming language should provide some facility for upgrading the old instances.

The literature promotes three possible ways of dealing with this problem:

Emulation: All interaction with old instances is via a set of filters, which support all the new-style operations on the old-style data format. Under this scheme, all of the information associated with the old type is retained, the filter functions masking their presence from the programmer. An additional feature here is that old programs can still run (on the old instances) without having to be repaired to handle the new type definition. [SZ87]

Eager Conversion: Write and execute a one-time program that iterates over all the instances of the type in the system, converting them into an instance of the new type via some user-specified constructor. For this approach to be possible, a system would have to support some method for retrieving all instances of a given type. It might also require a considerable "downtime" on the part of the database. [**BMO+89**]

Lazy Conversion: Whenever an old instance is found, automatically convert it to a new instance (using the same constructor mentioned under Eager Conversion). One complication associated with this scheme is whether or not to perform the conversion on when the old object is not being modified. Conversion of objects at read time would make read operations much more expensive. [BH89, **BMO+89**]

The creators of OPAL use the term **screening** to refer to both emulation and lazy conversion. From their perspective, emulation is a (very) lazy conversion operation, one that defers the actual conversion indefinitely.

Relational database systems have a number of characteristics that make the evolution of types (*i.e.,* the reformatting of tables) relatively easy. First, the structure of their types is relatively simple and uniform. Second, the table hierarchy is flat: tables are only defined at the top level, and the location of all instances (*i.e.,* relations) is known. As a result, table restructuring, or **schema evolution**, can be performed according to one of the first two procedures in a fairly painless manner.

Database programming languages, on the other hand, are not that fortunate. A number of common features of database programming languages interfere with the smooth integration of a type evolution scheme. In a statically-typed language, for instance, redefining a type will shadow the definition of the former type. Older programs, compiled under the previous type definition, will be able to operate on older instances, but will not recognize new instances. The converse will be true with programs compiled after the type redefinition. Type evolution, an inherently dynamic process, is at odds with the type system.

If a database programming language wishes to support one of the conversion methods outlined above, it will have to contend with another feature of modern programming languages, pointers and the nonuniform structure of data. Any correct conversion procedure would have to preserve all references to the object that is being transformed. This is particularly tricky if the conversion procedure results in a change of the (persistent) address of the object.

In spite of these problems, however, a few systems do make an attempt to explicitly deal with the issue of type evolution. [SZ87, **BMO+89**, BH89]

2.1.7 How is persistence implemented by the underlying system?

Rather than describe the implementation of a reviewed system in exact detail, I instead discuss any implementation feature that seems particularly clever, is relevant to a significant language-level feature, or provides support for efficiency-related features. Possible optimizations that the persistent object system might choose to support include:

Persistent Addresses and Page Faulting: Often, an implementation can take advantage of the features of the underlying operating system, and use hooks into the virtual memory manager to provide (interrupt-driven) automatic interning of persistent objects.

Swizzling: Once a persistent object has been **interned**, (*i.e.,* installed) in memory, one would like to minimize the expense of manipulating such objects. One such way is to overwrite all references to such objects with their (virtual) memory addresses, thereby avoiding future tests to see if the persistent object has already been interned. This scheme incurs an additional expense when writing objects back to the persistent store, as the persistent addresses must be recovered. [**RC89b**, Mos90]

Clustering: Some efficiency can be achieved by exploiting reference patterns on persistent data. Information about these patterns can be achieved in a number of ways. One method is to provide to the client the ability to pass hints about the locality of object references directly to the persistent object manager. A client could identify an existing object near which new objects should be created. Others methods are less dynamic, arranging the persistent data according to some basic assumption about access patterns. PS-Algol and E instantiate similarly-typed objects to common disk regions, under the assumption that object of the same type are frequently referenced together. [**ABC+83**, **RC89b**]

Lock Coalescence: The time spent locking shared objects could be reduced if the compiler were able to detect locking operations to neighboring objects (or parts of the same object) in the program, and lock larger regions of the store, instead of smaller regions in succession. This would be especially useful in array references, where a single lock on the entire array would be more efficient than locking each element in turn. [**RC89b**]

Sorting for specialized query processing: Often a system might try to optimize the arrangement of a data collection on secondary storage, so as to favor potentially common queries, such as searching for the record with a minimal attribute value, or locating all entries satisfying some equality constraint. [**BMO+89**]

Index tables: Specialized tables can be provided in order to support associative access on individual records. This approach is the basis of efficient database lookup in commercial relational database systems. [Dat81]

2.2 Related Language Issues

Some secondary but nevertheless interesting questions deal with aspects of the languages beyond persistence. It is worth investigating what sort of additional support is provided for database-type applications, and also, what the programming languages look like in general.

2.2.1 How does support for persistence integrate with the rest of the language?

Most of the database programming languages being developed are based on existing programming languages. The degree to which the language extensions are smoothly and consistently integrated into the base language is noteworthy.

It has already been mentioned (p.5) how the database-motivated additions to Pascal in Pascal/R are not universally applied. The more modern persistent and database programming languages are often subject to similar criticisms.

2.2.2 What database-oriented types and constructs have been provided?

The inclusion of database-oriented data types and control forms in a programming language offers potential advantages in both the functional and efficiency domains. Relational database systems manage their immense body of data with collections of like-typed data, providing aggregate and query/select operations and specialized control structures to operate over them. Likewise, PASCAL/R provides the `each` operator for mapping over `relations` in database files. (See p.5 for details.)

Commercially available database management systems provide highly efficient execution of some popular collection-oriented operations through the use of specialized data layouts and precomputed element addresses in the form of index tables. These index tables are created on demand by the database manager. Some database programming languages attempt to implement similar measures with minimal effect on functionality. For example, OPAL, rather than provide special data types, supports with a similar functionality via the use of optional type specifications: a programmer wishing to perform inexpensive searches over a collection of objects is encouraged to declare type restrictions on the relevant instance variables, allowing the system to produce code to iterate over the instance collection uniformly and rapidly.

2.2.3 Transaction Management

In a system where the data may persist for long periods of time, safeguards must be taken to promote the reliability of the store. Transaction systems make data more tolerant to failure, by grouping a sequence of operations into one atomic action. From the point of view of persistent languages, the important aspects of an underlying transaction system are:

- How transparent is the transaction system in the language's computation model?
- How sophisticated is the transaction model?

2.2.4 Concurrency and Locking

Large database systems should support concurrent access, and this applies equally well to the persistent programming language domain. language might choose to export some of the lock control to its users. Relevant questions include:

- How transparent is the locking facility from the programmer?
- What is the granularity of the locks?

2.3 Other Questions

I will include answers to the following questions when the authors choose to discuss them.

- Is there any provision for the security of objects in the persistent store? Can one application (or user) modify the data of another, without explicit permission? [*Just as operating systems protect programs from each other by partitioning memory, a complete database programming language system would have to include provision to exclude a malevolent application from damaging the store.*]
- What are the motivations behind the development of the language? What sort of applications are the authors trying to support? [*Usual responses are CAD/CAM, programming environments, graphic systems, and traditional database applications.*]
- What shortcomings are there in the language/system? [*If the authors cite any problems with their resulting language or system, it would be beneficial to repeat them.*]
- Does the system run? What do the users think? [*This is typically a hard question to answer, as clients do not regularly publish testimonials in journals, but sometimes some hints present themselves. For commercial systems, it is possible to find out something.*]

2.4 Structure of the Remaining Sections

Surveys of selected persistent programming languages follow in the remaining sections. The languages were selected according to present to the reader the variety of efforts in the field, with particular concentration on the variety of model of computation and persistence, base language, and focus.

At the end of each section, I include a short, annotated bibliography of the research papers published by the designers and implementation of the system being discussed. Papers are divided into four lists, as follows:

Primary Sources: Papers primary used in researching the survey. Readers interested in further information about the system are encouraged to refer to them. (References to these sources are cited in boldface in this paper.)

Secondary Sources: Papers read and found to contribute little information that cannot be found among the primary sources.

Other References: Papers that either focused on tangential aspects of the system (*i.e.,* not on persistence) or that I have failed to acquire and read.

Derivative Papers: Papers describing work derived from the featured system that might interest the reader.

3 PS-Algol and Napier

PS-Algol, an extension of S-Algol [Mor79, Col82]*, was designed and built by Atkinson et. al. between 1982 and '85 as part of the Persistent Programming Research Group's (PPRG) effort in Scotland. Its primary improvement over previous designs/systems was the notion of* **orthogonal persistence**. *It is generally acknowledged as the first persistent programming language.*

The successor to PS-Algol, Napier, adopts the same principles as PS-Algol, but attempts to reduce the prominence of dynamic type-checking in the language.

The two languages are very similar in their treatment of persistence. The following survey will focus on PS-Algol, while noting any distinguishing features of Napier, as relevant.

3.1 Issues

Model

The persistent store in PS-Algol is modeled as a collection of database files. These abstract files are opened explicitly, but while a database file is open, access to any persistent object stored in that file is transparent.

Multiple files can be opened for reading, but a program can only modify one database file at a time, so that the resident database file for new persistent data can be inferred. Modified and newly created objects are automatically moved back to the database at program completion, or at the explicit closing of the file by the user.

Pervasion and Declaration

PS-Algol was the first implementation of a language based on the principle of **orthogonal persistence**, which dictates that persistence is characteristic of a data value and not of its particular type. As a result, any object in PS-Algol can persist. The object returned by the `Open·database` routine is the **root** of the database, which, via successive pointer references, reaches all objects stored in that particular file. Similarly, any object reachable from this persistent root when the program completes (or when the database is explicitly closed) is written into the database file.

Addressing

PS-Algol supplies a new data type, the `table`, that supports associative access on objects based on keys of strings or integers. Each database is rooted at a `table` object, in order to make searching more amenable, but the user is free to use other composite data structures to arrange the database as he wishes. A uniform top level object allows different programs that share the database file to construct their own paths to objects.

In Napier, the `table` is replaced by the `environment`, which is a cross between a table and the static block, common to Algol-like languages. Like the table, entries (bindings) can be added or removed from the `environment`. However, like the static block, procedures can be closed under `environments`. All bindings to environment variables are resolved statically at closure time. Thus, procedures can be protected against operations that result in changes to the closing `environment`'s bindings.

Manipulation

Following one of the designers' driving principles, persistent type instances are of the same type as their volatile cousins, and can thus be operated on by the same methods.

Implementation

The automatic importing of objects from the persistent store is handled by the runtime system, which detects references to objects in the store (based on the sign bit of the address) at pointer de-reference time and copies the referenced object into active memory. Pointers to persistent objects that have been interned are **swizzled**, that is, overwritten with the local address, to reduce further trapping. Local addresses are converted back into persistent addresses when the data is written back to store.

In an effort to reduce the size of the working set, objects are arranged in the store according to their type. This way, programs iterating over homogeneous collections (a common operation for database applications) would touch fewer pages.

PS-Algol's specialized type for indexing, the `table`, is implemented using B-trees, for fast lookup and updating.

Integration

A major feature of the base language's type system is exploited to a significant degree by PS-Algol. In S-Algol, all user-defined data types are represented as tagged tuples, referenced by an untyped pointer. In order to verify type consistency, type checking is performed as part of the pointer dereference at runtime. PS-Algol takes advantage of this tagged architecture by extending it directly into persistent storage. Tagging of persistent data is essential for PS-Algol's implementation of orthogonal persistence, because it allows the system to automatically trace all references to persistent objects at transaction completion.

The dynamic type checking associated with this scheme is exploited at both the language and implementation levels. Persistent objects can only be referenced by untyped pointers, making the implementation of automatic persistent object interning easier (*c.f.,* previous section). However, this also means that the type check must be performed repeatedly throughout a computation. The authors point out that a certain amount of dynamic behavior in a persistent language is essential, because of the long-time, evolving nature of programming in a persistent environment. In Napier, the dynamic type-checking is localized by allowing programs to close over `environments` at runtime as well as at compile time. Strong type-checking is not sacrificed under this scheme, and static type-checking is employed everywhere else in the language. [**Dea89**]

The authors describe how the first-class procedures of S-Algol can be used to implement procedure libraries, modules, and abstract data types. This functionality has additional utility when combined with persistence. The act of importing an abstract data type from the persistent store and dynamically binding it into a program is equivalent to module linking in more traditional languages. The authors express excitement over the idea of integrating major portions of the program development cycle into the language system.

Napier features the addition of first-class processes. It is interesting to note that, in accordance with the property of orthogonal persistence, Napier processes can be preserved in the persistent store. A process in the store remains active; any application can export it and communicate with it. In this way, processes provide a convenient medium of communication for concurrent Napier applications.

Type Evolution

As theirs was early research, the authors of PS-Algol and Napier did not attempt to deal with type evolution. However, the PS-Algol implementation, in order to insure that the type information is available at runtime (for the dynamic check), stores the type description with the persistent object. This means that in the event of a type redefinition, instances of the old type will type-check correctly, and can be manipulated as before.

Database-oriented types

In PS-Algol, a set of functions is provided that can be used to apply some procedure over each entry in the `table`. No mention of a similar functionality is found in the Napier literature.

Transactions

The transaction model presented by PS-Algol is very simple. The act of opening a database file for writing begins a transaction, and the act of closing the file (either explicitly or by program termination) commits it. Routines to explicitly commit or abort the transaction are provided as well.

Concurrency and Sharing

Locking in PS-Algol and Napier is restricted to implicit, exclusive write locks on entire database files. More than one database can be opened for reading, but only one can be opened for writing. There is an additional restriction that a program can only reopened if it had not been written to by another program in the intervening time. This is so there are no inconsistencies between the database and any objects the program may have imported from the database when it had been previously opened.

The authors note that even a persistent store with such limited concurrency has uses. They cite as examples — CAD systems, where coarse locking granularity is not a problem, and for personal databases, where sharing is not necessary at all.

In Napier, the authors have included a notion of persistent process, which can be used to control concurrent access to persistent data, and to model more sophisticated transaction systems.

The most serious deficiency of PS-Algol relates to its locking behavior, which restricts a program to exclusive write access on a single database file at a time (i.e., *per transaction). So, while the coarse locking granularity would tend to encourage users to distribute their data across a large number of small database files (to increase concurrency), the lack of the ability to lock (and write to) more than one database per transaction promotes large database files.*

The utility of multiple database files is also hampered by the transparent nature in which objects are updated in the persistent databases. Even if it were possible to write to more than one database file simultaneously, cross-database references are not allowed. Unless that deficiency is repaired as well, it would remain impossible for an object to be shared among objects residing in different database files.

Napier's method of supporting both static typing and a restricted form of dynamic typing and binding is interesting. It reduces the amount of dynamic checking to a minimum, and does not sacrifice strong typing.

3.2 Sources

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisolm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983. Reprinted in ZdonikMaier90-Readings[ZM90].

A presentation of the PS-Algol language model, and touches on some of the issues of implementing persistence (issues which are elaborated elsewhere [ACCM83, ACC83, CAC+84]*).*

[AM85] Malcolm P. Atkinson and Ronald Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, October 1985.

Atkinson and Morrison discover that having first-class procedures within PS-Algol's persistence facility (as discussed in **[ABC+83]** *) provides the programmer with considerable power, including some higher-order programming language concepts, and perhaps enough even power to encompass the entire programming cycle within the PS-Algol language system.*

[Dea89] Alan Dearle. Environments: A flexible binding mechanism to support system evolution. In Shriver [Shr89], pages 46–55.

Focuses on the Napier data type `environment`*, a new type in Napier that is used to implement restricted dynamic binding.*

[MBC+88] R. Morrison, A.L. Brown, R. Carrick, R. Conner, and A. Dearle. On the integration of object-oriented and process-oriented computation in persistent environments. In Dittrich [Dit88], pages 334–339.

While providing a brief description of Napier, this paper emphasizes one new and interesting feature of Napier: the introduction of first-class processes.

Secondary Sources

[ACC82] Malcolm Atkinson, Ken Chilsholm, and Paul Cockshott. PS-Algol: An Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

The canonical PS-Algol reference, this paper introduces PS-Algol. Much of its contents are restated or improved in [**ABC+83**].

[AM84] Malcolm P. Atkinson and Ronald Morrison. Persistent first class procedures are enough. In *Proceedings of the Fourth Conference of Software Technology and Theoretical Computer Science*, volume 181 of *Lecture Notes in Computer Science*, pages 223–240. Springer-Verlag, Berlin, December 1984.

Describes how having first-class procedures and an embedded compiler can be used to implement abstract data types, data protection, and separate compilation. An early version of the information later presented in [**AM85**].

[CAC+84] W.P. Cockshott, M.P. Atkinson, K.J. Chilsholm, P.J. Bailey, and Morrison R. Persistent objected management system. *Software – Practice and Experience*, 14:49–71, 1984. Reprinted in ZdonikMaier90-Readings[ZM90].

Discusses the most recent implementation for the PS-Algol persistent object manager. Concentration on how shadow pages are used to implement (simple) atomic transactions; how structure instances are managed by type under the assumption that like instances are more often referred to by common transactions.

[MBC+89a] R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle, and M.P. Atkinson. The napier type system. In Rosenberg [Ros89], pages 253–269.

Discusses the Napier type system, and its reliance on restricted dynamic binding to aid in construction and modification of persistent programming applications.

Other References

[ACC83] M.P. Atkinson, K.J. Chilsholm, and W.P. Cockshott. CMS — a chunk management system. *Software – Practice and Experience*, 13:273–285, 1983.

Outlines the first implementation of the underlying persistent object manager for PS-Algol. Replaced by the system featured in [CAC+84].

[ACCM83] M.P. Atkinson, K.J. Chilsholm, W.P. Cockshott, and R.M. Marshall. Algorithms for a persistent heap. *Software – Practice and Experience*, 13:259–271, 1983.

An even more ancient implementation paper for PS-Algol. Only useful information is the admission that the implementors first thought about persistence as a way of running large processes on their small-memory computer.

[AM87] Malcolm P. Atkinson and Ronald Morrison. Polymorphic names, types, constancy and magic in a type secure persistent object store. In Carrick and Cooper [CC87], pages 1–12. Persistent Programming Research Report 44.

Outlines outmoded design of first-class environments and naming in Napier. Rendered obsolete by [**Dea89**].

[MBC+89b] R. Morrison, A.L. Brown, R. Carrick, R. Connor, A. Dearle, M.J. Livesey, C.J. Barter, and A.J. Hurst. Language-design issues in supporting process-oriented computation in persistent environments. In Shriver [Shr89], pages 736–744.

Features a discussion of addressing and first-class processes in a distributed Napier system.

3.2.1 Derivative Papers

I found a number of other papers which discussed work that used the PS-Algol system as a research platform. Cooper [CADA87] *describes how traditional database systems can be written in PS-Algol. Dearle and Browne* [DB88] *illustrate how a type-safe object browser can be constructed using dynamic typing and a built-in compiler. Philbrow* [Phi87] *replaces the PS-Algol* `table` *object with a more powerful indexed structure, derived from traditional database systems. Wai* [Wai87, Wai89] *extends the PS-Algol implementation to support distributed access, retaining the computation model as much as possible.*

4 Amber

Amber is a language developed for the Macintosh by Cardelli. An acknowledged ML spin-off, it includes support for graphics, concurrency, and persistence. Amber promotes a very simple view of the persistent store, and makes use of the **dynamic** *type* [ACPP89] *to interact with it.*

4.1 Issues

Model

Amber offers a very simple model of persistence, viewing the persistent store as no more than an archival storage medium for objects. The store is no more part of the language than the file system. The `export` routine takes a value and a string name, saving the value for later retrieval under the string. The `import` routine, given a string, retrieves the value stored under that name.

These operations are sophisticated enough to support sharing of sub-objects within an object, so any internal references are preserved. However, the sharing of references among persistent values (or even successive imports of the same object) is not supported. This lack of identity between volatile and persistent representations is the essence of Amber's simplicity with respect to persistence.

Pervasion and Declaration

In Amber, any object can be converted into an instance of the `Dynamic` type, which combines the value and type parts of the object. A `Dynamic` instance can be later coerced back into a copy of the original object, given its original type as an argument. The operation is perfectly type-safe, as the `coerce` operation performs the required type-checking at runtime, signalling an error if the value being coerced is not of the supplied type. The semantics of `Dynamic` are similar to the CLU [LAB+81] `force` type generator. Cardelli *et. al.* describe the **dynamic** data type more formally in a recent paper. [ACPP89]

Only objects of type `Dynamic` can be written to store, but since all types are coercible to and from `Dynamic`, there is effectively no restriction on what types of values can persist.

Persistence in Amber is not a property that can be declared, but rather, a user must explicitly write a value out into the store for it to persist.

Addressing and Manipulation

Persistent values are uniquely identified with the string they are stored with. `Import` takes a string as its argument, so one can "refer" to a persistent object via its (string) filename.

Amber provides no support for the direct manipulation of persistent values. Such values must first be read and coerced into volatile, typed objects before any computation can be performed. The actual changes that are performed on these objects will not persist unless explicitly written back to store.

Type Evolution

Neither paper discusses the problem of type evolution, but the following conclusion can be made based on Amber's characteristics: in order to coerce a dynamic instance back into original type, one requires the type. As part of the coercion operation, the type provided is compared with the type description that was encoded with the instance value when the dynamic value was first created. As a result, if a type is redefined, all its former instances in the persistent store, while still accessible, are no long usable, since they cannot be converted from their dynamic representations.

Implementation

The persistent store is implemented directly on top of the underlying file system. The string required by the read and write operations are used to generate a file name, under which the value's "persistent" representation is saved. A new file is generated for each object, and reusing a file name overwrites the previous persistent value contained there. A special representation for `Dynamic` instances in ASCII (!) is defined, with the `import` and `export` routines converting between the external and internal representations.

Additional Comments

Amber was designed to run in a single-user environment, and thus has no support for the sharing of persistent data. It also makes no effort to support database integrity at the system level.

Dynamic type checking on objects in the persistent store is done explicitly as a result of the `Dynamic` *instance coercion. This avoids the problem experienced by PS-Algol, which was subjected to pervasive dynamic type checking. However, the lack of identity between persistent and volatile versions of an object presents an nonetheless primitive model of persistence. Having been designed for a (single-user) Macintosh, Amber offers no support for concurrent access or locking.*

4.2 Sources

[Car86a] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 48–70. Springer-Verlag, Berlin, 1986.

This paper presents an overview of the Amber system.

[Car86b] Luca Cardelli. The Amber machine. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, Berlin, 1986.

This paper defines the abstract Amber Machine. In addition to providing a description of all the Amber Machine operations, a number of implementation issues relevant to persistence are mentioned.

5 Galileo

Galileo, an interactive programming language for database applications, was designed by Albano and Orsini (and Cardelli) at the University of Pisa. Influenced by previous work in persistent programming languages, notably PS-Algol [**ABC+83**] *and efforts in the database world* [BZ81, MK84], *it was specifically designed to support both progressive database modeling* and *the abstraction mechanisms of current programming languages. Galileo's most interesting feature is its type system, which provides specialized features for database-like operations while remaining useful under traditional programming language paradigms.*

5.1 Issues

Model, Pervasion and Declaration

In Galileo, all objects persist indefinitely; objects that are not referenced are removed by the built-in garbage collector. This results in a model which presents the entire persistent store as an active heap. Objects are swapped into active memory and back out to the store as required, much like a virtual memory system. In fact, the authors imply that while they have no interest in adapting a virtual memory system to handle the Galileo store, that course is likely the preferable way to accomplish their goal.

Addressing

The facilities for naming and associating objects in the Galileo store are drawn from the two domains that inspired the language. Similar to traditional programming languages, Galileo provides the `environment`, a statically-scoped block mapping names to objects. From the database (and perhaps object-oriented) world, Galileo provides aggregate data structures. Programmers have the option of associating with any of their class definitions a sequence that is defined to contain all the (existing) instances of that class. Various kinds of database-like queries can be performed on these instance collections, resulting in an effective addressing mechanism for the class instances.

Type Evolution

Galileo does not address the issue of type evolution. As it is a statically-typed language, old instances can be manipulated by functions if they were compiled when the instances' type definition was accessible by name.

Implementation

Unlike PS-Algol, which converts persistent addresses into local ones when the referenced object is fetched from the store, the Galileo implementation maintains persistent addresses throughout. This attitude enforces the analogy between virtual memory and the Galileo persistent store.

The implementation does include one significant optimization with respect to persistent address translation. Taking advantage of its generational garbage collection, the implementation ensures that the youngest GC **generation** is always in active memory, and that within it, the persistent (virtual) address corresponds directly to the local (physical) address. This strategy relies on the notion that newly-created data is short-lived. The authors note that this is not likely to be true for functional data, and provide no method for coping with that property.

Database-Oriented Types

The instance collections for classes, mentioned previously (*c.f.,* **Addressing**), are but one example of the database-like support provided by Galileo. Other useful characteristics of class definition that are inspired by existing database language facilities are:

Key Attributes: A special **key** can be declared to force the values of a subset of an instance's attributes to be unique among its co-instances. [*This would allow an implementation to support indexing over the class, using the key attributes as effective keys, similar to the facilities common to relational database systems.*]

Derived Attributes: An attribute value that is recomputed from primitive values every time it is referred to. [*While functionally similar to a function of no arguments, derived attributes can be read as regular attributes. Some database systems enhance this feature by allowing an inverse function to be defined, so as to support assignment.*]

Subtyping: The ability to define subclasses of a class. Instances appear in the instance sequences of both class and superclass.

Transactions

Galileo is an interactive system. Any expression evaluated at the top-level is executed as a simple transaction. Compound transactions can be performed by using the `transaction` and `end_transaction` forms, which identify a sequence of commands to be performed atomically.

The only way to abort a transaction is to raise an exception. The exception handling system can be used to perform cleanup operations at abort time. Exceptions are distinguished by string names.

Transactions can be nested by simply composing operations. Each function call is run as a nested transaction, and any exceptions can be caught locally. If the top-level call raises an exception, the effects of the entire series of nested transactions are undone.

Concurrency and Sharing

The authors are suspiciously quiet about Galileo's locking facilities, leading me to conclude that there is no support for concurrent database access.

In spite of the current design of the system which ignores the issue of concurrent access, it is possible to extend the language description to provide such support. In keeping with the feature of total transparency of persistent data access, such an extension should avoid explicit locking operations. However, implicit locking at the granularity of objects could be provided, with conflicts and deadlocks resolved by automatically generated transaction aborts.

5.2 Sources

[ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985. Reprinted in ZdonikMaier90-Readings[ZM90].

Introduces Galileo, and provides an overview of its interesting type system, and of its first-class environments.

[AGOP88] A. Albano, F. Giannotti, R. Orsini, and D Pedreschi. The type system of Galileo. In Atkinson and Buneman [AB88], chapter 8, pages 101–199. *Based on the proceedings of the First Appin Workshop, which appeared as PPRG Persistent Programming Research Report 16.*

Similar in focus to [**ACO85**], *but with a heavier concentration on the semantics of its types.*

Other References

[Ghe85] G. Ghelli. La gestione della persistenza dei valori nel linguaggio Galileo. *Scienze dell'Informazione*, 1985.

A detailed explanation of Galileo's persistent Garbage Collector, apparently in Italian.

6 EXODUS and E

*EXODUS, a DBMS project from the University of Wisconsin, sets out to design a platform on which database programmers can quickly design and create fast, application-specific, database systems. Central to the system are the Storage Manager, which implements an efficient (albeit primitive) persistent store, and the programming language E, which provides application programmers a high-level interface to persistence data management. While its primary goal is not to provide a programming language for database, E is interesting enough to be presented here as a persistent programming language. E is one of many object-oriented database language efforts that are prevalent today. OPAL (*c.f.*, Section 7) is another.*

E is an upwardly-compatible extension to C++ [Str86], supporting parameterized data type constructors and iterators, as in CLU [LAB+81], and persistent type instances. It is designed to be used by database system implementors for database systems programming.

6.1 Issues

Model, Pervasion and Declaration

From the perspective of E applications, the persistent store is an extension of the C++ dataspace. Interaction with the store is transparent, except for the following, important characteristic. In the interests of preserving upward compatibility with C++, and in order to reduce the expense of manipulating volatile objects, the designers of E opt to sacrifice orthogonal persistence. Instead, two sets of types and type constructors are provided. Persistent types are declared and defined using the `dbclass` form. While instances of regular C++ types can reside only in local memory, instances of `dbclass` definitions can reside in either local memory and the store. In this way, the system can avoid having to test for the residency of an object if it is of volatile type. The authors note that it is possible to write E programs where all types are persistent (using only the persistent base classes and declaring all new classes as persistent), but warn of the serious runtime expense involved.

Addressing

Persistent bindings are handled rather elegantly. E provides a new variable declaration, `persistent`, which resembles a traditional C `static` binding, with the extension that the binding survives program executions in addition to successive procedure calls. [*See* **Integration** *section below for some associated problems.*] Typed aggregates of objects, called `collections`, can also be created to contain dynamically created instances of persistent types.

Manipulation

Once in active memory, persistent objects can be manipulated similarly to their volatile cousins. But since they are members of a different class hierarchy, distinct though analogous methods must be defined for them. Users can operate over elements of a persistent `collection` using iterators.

Type Evolution

No solution for the problem of schema evolution (*c.f.,* Section 2.1.6) is provided. Emulation is rejected by the authors, who claim that the addition of a layer between the EXODUS Storage Manager and the E program would seriously reduce efficiency. Automatic conversion, whether lazy or eager, is also rejected, as it does not mesh well with the C++ data layout. To implement immediate references to other classes and structures, C++ embeds class and structure instances within its referent. The resulting change in the size of the object might invalidate remote pointer references.

Implementation

One feature of the E implementation that distinguishes it from other systems described in this paper is how persistent objects are maintained and addressed. Like the PS-Algol implementation, E has two pointer types, one for its C++ type instances, and one for its persistent types. Unlike PS-Algol, the persistent pointer is represented as a pair of values: a persistent object id (similar to the persistent addresses we have seen in other systems), and an offset pointer. In this way, pointers into the middle of objects (a common occurrence in C idiom) is possible. Volatile instances of persistent types are represented by a special persistent id and an offset corresponding to the object's main memory address.

An early decision in the design of the E implementation was to use the C storage model for both volatile and persistent data, so as to avoid any format conversion. This was done for additional efficiency, but has resulted in a serious problem. Unlike other systems, such as PS-Algol, which used pointer swizzling to reduce demand on the store (*c.f.,* p.17), the E compiler might generate code that would repeatedly lock and intern an object from the store. The implementors hope that an adaptation of register allocation techniques to this problem will alleviate its affects in future releases.

A second side-effect of using the same storage model for volatile and persistent data relates to object deallocation. E provides no garbage collection operation on the persistent store, exposing the programmer to a familiar pair of potential bugs. Any garbage generated through missing deallocation operations will persist indefinitely. Likewise, objects might be accidently deleted prematurely, resulting in a corrupt database.

The underlying persistent object manager has some optimization features as well. At object creation, the EXODUS object manager allows the client to specify an existing object near which the new object should be created (if possible). This operation is a way of enforcing locality at the database design level.

In order to efficiency support E's `collections`, EXODUS provides a data structure known as the **file object**, which is designed to operate effectively under both direct *and* sequential access patterns.

Integration

E has yet to resolve some problems related to the management of its persistent namespace, mostly resulting from the fact that compiled E programs reside outside of the persistent store. The most serious problems involve the execution of destructors on (static) `persistent` instances, and the handling of virtual functions. In the current system, a special and clumsy utility program is provided to allow programmers to delete `persistent` bindings when an associated program module was being retired. And in a world where each compiled program has different addresses for a dbclass's virtual functions, the dispatch table must be a volatile object. Both of these problems will be avoided in the future implementation, which will see programs and program modules inserted into the persistent store at compilation.

Database-oriented types

The principal database-oriented type provided by E is the `collection`, which has been mentioned previously. Operations on instances residing in the `collections` can be performed using the specially provided `iterator` control form. However, as EXODUS does not support any form of indexing on `collections`, their elements can only be operated upon sequentially.

Transactions

Currently, there is no explicit transaction facility; each program runs as its own, simple transaction. Future implementations will support some level of programmer control over transactions via a language-level, `transaction` block.

E is unique among the languages featured in this survey in that it extends a language that lacks a tagged data architecture. In this environment, previous approaches to type evolution are no longer applicable, and the language's creators have yet to propose a solution of their own. Another trait of an untagged architecture is the lack of a garbage collector for the persistent store. This forces the programmer to manage allocation and deallocation of persistent object explicitly, and subjects the store to a number of errors that can damage its integrity.

The current implementation of E requires the presence of an external utility to help manage the relationship between persistent bindings and module implementations. This lack of integration is clumsy, but future implementations will have code reside in the store, removing the need for external manipulation of the database.

The lacks of transactions is another shortcoming of the current implementation, but the creators express a desire to incorporate a transaction system in their next implementation.

6.2 Sources

[RC89b] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software – Practice and Experience*, 19(12):1115–1150, December 1989.

A discussion the database programming language E, constructed as part of the EXODUS Extensible DBMS Project. It concerns itself solely with the design and implementation features of persistence in E.

Secondary Source

[CDRS86] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Skekita. Object and file management in the EXODUS extensible database system. In Yahiko Kambayash, editor, *Proceedings of the Twelveth International Conference on Very Large Data Bases*, pages 91–100, Kyoto, August 1986. Very Large Data Base Endowment.

A description of the design of the EXODUS Storage Manager. In order to support a large variety of database applications, the manager was designed to be as simple and flexible as possible.

Other References

[RCDS87] J.E. Richardson, M.J. Carey, Dewitt D.J., and D.T. Schuh. Persistence in EXODUS. In Carrick and Cooper [CC87], pages 96–113. Persistent Programming Research Report 44.

At the Second Appin Conference, the EXODUS group presented their language model for persistence, briefly describing the Storage Manager and introducing E, the persistent programming language. Most of the issues presented here reappear in later papers, notably [**RC89b**].

[RC87] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In U. Dayal and I. Traiger, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 208–219, San Francisco, CA, May 1987.

Similar in subject to [RCDS87]*, but from the point-of-view of a database implementor. Focuses on the problems facing the designer of a database application, and how the facilities of E assume much of the burden.*

[CDV88] Michael J. Carey, David J. DeWitt, and Scott L. Vandenberg. A data model and query language for EXODUS. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 413–423, Chicago, IL, June 1988. *Also available as WISC-CS-TR 734.*

A presentation of a data model (named EXTRA) and a query language (called EXCESS), built on top of E, explicitly to illustrate how database models can be easily constructed on top of EXODUS.

[CDG+90] Michael J. Carey, David J. DeWitt, Goetz Graefe, David M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Skekita, and Scott L. Vandenberg. The EXODUS extensible DBMS project: An overview. In Zdonik and Maier [ZM90]. *Also available as WISC-CS-TR 808.*

A formal overview of the entire EXODUS project, describing in summary the major aspects of the project (including the Object Manager, E, the sample data model presented in [CDV88]*, and a query optimization system.*

[RC89a] Joel E. Richardson and Michael J. Carey. Implementing persistence in e. In Rosenberg [Ros89], pages 302–319. *Republished as part of* [**RC89b**].

A presentation of the current implementation of the E compiler. Republished as part of [**RC89b**].

7 GemStone and OPAL

The GemStone system is a commercial implementation of an distributed, object-oriented database system developed at Servio Logic Corp. The system consists of the GemStone Object Server (an object-oriented database server), the OPAL programming language (a Smalltalk-80 extension) for writing database applications, and restricted server interfaces for C and PASCAL, which are used for user-interface application construction. This survey concentrates on OPAL.

The GemStone system is implemented as a distributed system: a single, central database server surrounded by both smart (*i.e.,* workstation) and dumb (*i.e.,* IBM-PCs) clients. The central server runs the persistent object manager (Stone) and the session manager (Gem) if the client is not powerful enough to run it itself.

7.1 Issues

Model, Pervasion, and Declaration

OPAL presents a single persistent store to the user, and access to persistent objects can be transparent. However, programmers interested in dealing with concurrency, security and integrity issues are provided (explicit) facilities to do so. (See **Additional Comments** heading, below.)

OPAL extends the Smalltalk heap into the persistent store. All types and instances persist indefinitely; an object persists as long as a reference to it exists. A special garbage collection scheme is used to manage the store.

Addressing

Naming in OPAL is provided at two levels. Similar to the standard Smalltalk run-time environment, each user maintains his own (persistent) namespace for resolving bindings. In addition, OPAL supports collections of class instances and primitive operations on them, similar to Galileo classes, but under explicit user control. (More on these collections below.)

Type Evolution

The authors reject the emulation scheme and the lazy conversion approach as previously outlined. Instead, they favor a mixed strategy, which involves lazy conversion until the next garbage collection, at which point all remaining old instances are upgraded. (Their current implementation, however, does not yet support this feature — the conversion being done eagerly for the time being.) They identify a list of constraints which must be preserved across modification to type descriptions and to the inheritance hierarchy. The authors then proceed to enumerate a number of categories of object updates that are permitted, and what changes to the dependent instances and subclasses must be performed in order to maintain the integrity of the database (*i.e.,* to preserve the above constraints).

Integration

OPAL allows type restrictions to be assigned to instance variables. In addition to enforcing integrity within the database, such **class-kind** declarations can be used to identify optimizable slot references, called **path expressions**.

Three types of composite types are provided by OPAL: indexed arrays, named records, and unordered collections, represented by the built-in types: `Set` and `Bag`. The `Array` class supports insertion and deletion of elements, and (numerically) indexed lookup. Record fields are fixed in number, but are addressed by name. The unordered collection classes support the standard set operations, and an optimized form of query processing (described below). Each of these three types of composite object is essential to an object-oriented database programming language, which requires structures that are suitable to both the database and programming language paradigms. (Smalltalk only supports arrays and records, while database languages typically support only records and unordered collections.)

Database-oriented types

OPAL supports indexed access on the values of path expressions over unordered, homogeneous collections of instances. Because of the class-kind restrictions, the operation to access the slot value is uniform over all the instances in the collection. [*In this way, OPAL supports the query operation with relative efficiency, and follows a procedure that is consistent with the base language.*]

This query optimization scheme is made even more useful by the presence of subclasses in the language. Instances of a subclass can appear wherever a class instance is required, and class-kind restrictions on inherited instance variables can be refined. However, the types of tests in the queries are restricted to identity and the so-called **equality** operations ($=, <, >$, *etc.*) that are defined for that instance type.

Transactions, Concurrency and Sharing

The GemStone designers recognized the need for supporting both optimistic and pessimistic concurrency control. Pessimistic locking is preferred for long-lived transactions, while optimistic locking is desirable in most other cases, as it favors readers over writers, and is (mostly) transparent to users. The authors note that since pessimistic control can easily be built on top of an optimistic scheme, it is better to implement the optimistic locks at the primitive level, and to support pessimistic control on top.

For each transaction, a list of objects that have been read or written is maintained. At commit-time, the list is compared with the list of objects that have been modified since the transaction began. If there are no conflicts, the transaction can commit, otherwise it aborts. All this occurs automatically; there is no need for user intervention.

Additional Comments

The persistent store is partitioned into **segments**, which are units of ownership and authorization. Users can create any number of segments, and can grant read or write permissions to them on a user or group basis.

Objects can refer to other objects across segment boundaries. So, even though a user might be able to access an object, he might not have the same permissions on its associated values and methods. This facility can be used to the database programmer's advantage to provide restricted views of objects to users and applications.

While providing a useful form of query optimization, OPAL's scheme is far from complete, and, in fact contains some of the restrictions characteristic of the database systems it attempts to replace. The restrictions on the types of tests possible within the queries (equality and ordering operations) appear to exist in order to allow the system to presort the collection according to the ordering relations. It would be better, however, if the system could be extended to support arbitrary predicates as filters for the query. While hardly as efficient as the preordering implied by the equality operations, there would still be some expected performance improvement resulting from the uniform data access paths.

7.2 Sources

[BMO+89] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The GemStone data management system. In Kim and Lochovsky [KL89], chapter 12.

A general discussion of GemStone design issues, with particular focus on how to maintain database integrity in the presence of schema evolution of class descriptions.

Secondary Sources

[MS87] David Maier and Jacob Stein. Development and implementation of an object-oriented DBMS. In Shriver and Wegner [SW87], pages 355–392.

Discussion of GemStone implementation issues, most of which are repeated in the later paper [**BMO+89**]. *An early version of this paper appeared as* [MSOP86].

[PSM87b] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an object server with other worlds. *ACM Transactions on Office Information Systems*, 5(1):27–47, January 1987.

Explains how client programs for the Smalltalk-based system can be written in Pascal and C on IBM-PCs. Illustrates the heterogeneity of the GemStone system.

[PS87] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In OOPSLA87 [OOP87], pages 111–117.

Earlier version of [**BMO+89**]. *Contains most of the same points, with some problems left unresolved until the later paper.*

Other References

[PSM87a] D. J. Penney, J. Stein, and D. Maier. Is the disk half full or half empty? *combining optimistic and pessimistic concurrency mechanisms in a shared persistent object base.* In Carrick and Cooper [CC87], pages 337–345. Persistent Programming Research Report 44.

Explains implementation strategy employed by GemStone group for supporting both optimistic and pessimistic locking protocols, for short- and long-term locks, respectively.

[MS86] David Maier and Jacob Stein. Indexing in an object-oriented DBMS. In Klaus R. Dittrich and Umeshwar. Dayal, editors, *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, pages 444–452, Pacific Grove, CA, September 1986. Institute of Electrical and Electronic Engineers.

[MSOP86] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. In OOPSLA86 [OOP86], pages 472–482.

An early version of [MS87].

8 Avalon/C++

Avalon/C++ is a language system providing support for distributed, fault-tolerant applications. A limited form of data persistence is achieved as part of the system's support for recovery in the presence of failure. Avalon/C++ (as well as Avalon/Common Lisp, featured below), makes use of the locally available Camelot transaction system [SBD+86] *for underlying system support, and a number of the language design decisions are motivated by the Camelot semantics.*

Like E (p.29), Avalon/C++ uses C++ as its base language. However, while E is mostly concerned with persistence, Avalon/C++ is primarily concerned with distribution and fault-tolerance. So, while many similarities exist between the two systems, some differences are noteworthy as well.

8.1 Issues

Model

Avalon/C++'s principle extension to C++ is the `server` form. Nearly identical syntactically to C++'s `class`, a `server` declaration establishes the interface to a server, an independent process running in its own address space, possibly on a different processor. Communication is via a remote procedure call interface to the servers exported functions. These functions are called normally, but their arguments (and return value) are passed under a copy semantics: only copies of their values are transmitted.

Avalon/C++ servers, once started, can persist indefinitely. Support is provided by the underlying system to restart the server process and restore relevant portions of its state in the event of a failure. This functionality provides the programmer with a limited model of persistence, allowing data to survive so long as the server is existent and its internal data organization remains constant. Another restriction, not characteristic of other systems we have seen, is that the persistent objects cannot be shared among multiple applications. While many different clients to the server can be written, those clients cannot get direct access to the data, which is protected by the server.

The "persistent store" in Avalon/C++ can be viewed as being distributed across multiple servers. The system requires clients to explicitly connect to the servers before attempting communication with them. Clients have the power to initiate execution of servers as well.

Pervasion, Declaration, Manipulation and Naming

A built-in persistent data type, `recoverable`, is provided. Programmers can create persistent types by defining classes that inherit from `recoverable`. This scheme establishes a dual hierarchy of volatile and persistent classes, similar to the one promoted by E. Instances of such persistent types can only be created by a `server` process.

Bindings of variable names to persistent values is achieved using the `stable` declaration. This is identical to E's `persistent` form, except that the `stable` declaration is restricted to the private members of `server`s.

As in E, persistent objects are members of their own class hierarchy, and can be manipulated by methods for those classes.

Implementation

Avalon/C++ uses the Camelot system to provide support for data persistence and distributed transaction management. Camelot provides a simple, uniform view of the persistent store (contiguous sequences of bytes), so no special data organization optimizations are possible.

Database-oriented types

No built-in database-oriented types or constructs are provided. E-style iterators could be built out of underlying C++ [Str86, p.183], but there would no performance improvement, since there is no underlying primitive system support for that operation.

Type Evolution

Avalon/C++ makes no accommodation for the redefinition of persistent data types in the server. Moreover, various characteristics of the language system make modifications to type (and server) definitions quite inconvenient. When the signature of a server is modified, for example, it is important to recompile both the client and the server. Regrettably, this constraint cannot be enforced by the language system itself.

Since persistent data is associated with the active server process, it is not possible to install a new version of the server (*i.e.,* code and type definitions) and preserve the associated (persistent) data of the replaced version.

Transactions

Camelot supports a number of advanced transaction features, including nested and distributed transactions. This functionality is exported to Avalon/C++ users. Syntactic forms are provided to create, commit, and abort nested and top-level transactions. Distributed transactions are supported transparently by the system.

Concurrency and Sharing

Light-weight concurrency is implicit within servers. Every server request, *i.e.,* call to server (method) function from a client, is run as its own process. Multiple light-weight processes can also be explicitly initiated using the `costart` form. Optimistic and pessimistic locking protocols (atomic and dynamic atomicity) are supported.

Data persistence is provided, but only in concert with a notion of persistent process. Data associated with a server persists as long as the server instance does. If the server is replaced (by a revised version), the associated data is lost. Unlike the persistent programming languages discussed previously, the data persisting in Avalon/C++ is protected by the server, and can thus not be directly shared by multiple applications. By contrast, persistent values can be shared by applications in E, which does not partition the dataspace.

8.2 Sources

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*. IEEE, July 1987.

[DHW88] David Detlefs, Maurice P. Herlihy, and Jeannette M. Wing. Inheritance of Synchronization/Recovery Properties in Avalon/C++. *Computer*, 21(12), December 1988.

[WHC+91] Jeannette Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietske, Richard Lerner, and Su Yuen Ling. The Avalon language. In Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors, *Camelot and Avalon: A Distributed Transaction Facility*, The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo, California, February 1991.

Other References

[WHC+88] Jeannette Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietske, Richard Lerner, and Su Yuen Ling. The Avalon/C++ programming language (version 0). Technical Report CMU-CS-88-209, Carnegie Mellon University School of Computer Science, December 1988.

Technical report version of material found in [**WHC+91**].

9 Avalon/Common Lisp

Avalon/Common Lisp is a prototype system developed to investigate how some of the characteristic features of Avalon/C++ could be integrated into a drastically different base language, such as Common Lisp [Ste90]. *It is included here because of my intimate association with the system.*

9.1 Issues

Model, Pervasion and Manipulation

Avalon/Common Lisp presents the following model of computation. The system is made up by a distributed set of **evaluators**, each a distinct Common Lisp process. Users can establish connections between their initial evaluator and others, and pass expressions over these connections, where they will be evaluated by the remote evaluation service. Many evaluators have an associated persistent store, on which may be stored objects that are guaranteed to survive hardware failure.

Avalon/Common Lisp provides transparent access to the objects in the persistent store; there are no explicit routines to import or export objects. Although the initial design of the system called for uniform manipulation of volatile and persistent instances, this was never achieved. Operations for the persistent base types (*e.g.,* `pairs` and `vectors`) are provided, as well as for persistent instances of user-defined `structures`.

Almost all of the Common Lisp data types admit persistent instances, the most notable exception being the `function` type. As a result, persistent code is not supported.

In a manner reminiscent of Amber (p. 23), universal operators are provided that create a volatile data type instance from its corresponding persistent instance, and *vice versa*. These operations have copy semantics.

Declaration and Addressing

Similar to E and Avalon/C++, Avalon/Common Lisp provides a construct to declare persistent variable bindings. The form, `defpersistent`, is an extension of the Common Lisp `defvar` form, declaring the variable and providing a default binding if a previous (persistent) value is not present.

Implementation

Like Avalon/C++, Avalon/Common Lisp relies on the Camelot system for persistent and transaction management. The persistent heap is managed by a Camelot server. This design made Avalon/Common Lisp operations on the persistent store particularly expensive, an acknowledged serious deficiency of the system. [CLNW90b]

Transactions, Concurrency and Sharing

The underlying support for distributed, nested transactions that is supported by the Camelot system is provided to the Avalon/Common Lisp user in an explicit manner. Special forms are supplied to begin new or nested transactions. As in Galileo, normal evaluation of the transaction expression commits the operation, while an abnormal termination of the subcomputation, either by explicit transaction abort or Common Lisp exception, results in an abort of the operation.

Avalon/Common Lisp was designed as a distributed programming system, but the realization of this goal was limited in a number of serious ways. The lack of lightweight concurrency within the base Common Lisp implementation prevented the Avalon/Common Lisp designers from supporting local concurrency.

Additional Comments

Avalon/Common Lisp's most distinguishing feature is its support for **remote evaluation**, which allowed the migration of subcomputations to non-local servers. The transmission of all Common Lisp types to remote evaluators is supported, and sharing within objects passed as part of the remote evaluation expression is preserved. Although similar to remote procedure call, remote evaluation provides the ability to coalesce a number of RPC invocations into one.

The lack of concurrency makes Avalon/Common Lisp practically unusable. Camelot was also a problem: its data model, which is C-based, is quite different from that required by Common Lisp, making access to the persistent store quote expensive, and making garbage collection of the store impractical.

The unification of the type system (i.e., *unifying volatile and persistent types) could have been achieved eventually, removing a considerable amount of the awkwardness present in the current implementation.*

9.2 Sources

[CLNW90a] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing. Reliable distributed computing with Avalon/Common Lisp. In *Proceedings of the International Conference on Computer Languages*, New Orleans, LA, March 1990. Institute of Electrical and Electronic Engineers Computer Society. *Also available as Carnegie Mellon School of Computer Science Tech Report # CMU-CS-89-186; also an extended abstract appears as "An overview of Avalon/Common Lisp," in the Proceedings of the Third Workshop on Large Grained Parallel Programming (Pittsburgh, PA, October 10-11, 1989).*

An overview of the features and characteristics of Avalon/Common Lisp.

Secondary Sources

[Cla89] Stewart M. Clamen. Towards Avalon/Common Lisp: Remote Lisp evaluation. Avalon Note 14, Carnegie Mellon University School of Computer Science, 1989.

Introduces and defines the concept of remote evaluation for Avalon/Common Lisp.

[CLNW90c] Stewart M. Clamen, Linda D. Leibengood, Scott N. Nettles, and Jeannette M. Wing. A programmer's guide to Avalon/Common Lisp. Avalon Note 15, Carnegie Mellon University School of Computer Science, 1990.

Describes the features and programming idioms for Avalon/Common Lisp.

[CLNW90b] Stewart M. Clamen, Linda D. Leibengood, Scott N. Nettles, and Jeannette M. Wing. Assessment of the Avalon/Common Lisp implementation. Avalon Note 16, Carnegie Mellon University School of Computer Science, 1990.

An Avalon/Common Lisp post-mortem, analyzing the final state of the prototype system.

10 Noteworthy Projects and Publications

10.1 Related Work

Below are brief descriptions of some additional persistent programming and systems. They were not included in the main body of the survey for one of two reasons: either they did not differ significantly from the highlighted languages, or there was a lack of research material available.

Taxis

Supports semantic data modeling using type inheritance. Represents an attempt at providing richer data semantics without resorting to an applicative programming language. [MBW80]

Modula/R

Successor to Pascal/R. [AB87]

DBPL

Successor to Modula/R. [SM83]

PLAIN

Similar to Pascal/R but with some additional features, including exception handling, aggregate operations over sets or sequences in addition to relations (unlike Pascal/R which restricted aggregate operations to relations.) [WSK+81, AB87]

Argus

A CLU-based language developed by Liskov *et. al.* for reliable, distributed applications. Distribution is based on the client-server model, using remote procedure calls for communication, and persistence takes the form of resilient data and bindings in the presence of hardware failure. As in Avalon/C++, however, data is wedded to the server it is associated with: data cannot be passed out by a server (server handlers are call-by-value), and persists only while the server is existent. [LDH+87]

In an effort to cope with the need for schema evolution, Bloom developed a system for replacing modules and transforming persistent data in Argus. [Blo83]

OM

A persistent object system extension to T (a Yale dialect of Scheme), running on top of the Apollo DOMAIN system, done as a doctoral thesis project. The data model is centered around the persistent heap, a new type of object in which objects can be created and manipulated. The author is primarily concerned with accessibility and ease of programming, and as a result, ignores all reliability issues. Concurrent access to heaps is supported by explicitly importing the DOMAIN system synchronization routines into T, and expecting the programmer to handle possible conflicts himself. [Mis84]

Poly and Poly/ML

Developed at Cambridge by Matthews [Mat85, Mat86], Poly is a general purpose programming language, with an execution model similar to ML, but with a different type system. The Poly/ML systems supports both Poly and ML "under the same roof", the two languages sharing a compiler back-end. Poly was initially designed to support persistence, and the Poly/ML system supports a persistent programming environment with an emphasis on transparency. The result is a system similar to that of Galileo [**ACO85**], but with some of the restrictions of PS-Algol [**ABC⁺83**]. Poly is an interactive language, and, like Galileo, the entire workspace of the user persists across sessions. Unreferenced objects are eventually garbage collected. Modified (and new) objects are written back into the database at the end of the interactive session (or when the `commit` operation is called) . In a method similar to PS-Algol, the database is divided at the abstract level into files. One significant improvement over PS-Algol is that cross-file references are supported. However, Poly still possesses one of PS-Algol's major shortcomings: primitive support for concurrent database access. There are no transactions; instead, shared database files are opened read-only, and any modification to the objects retrieved from that file do not persist. [*This is where the principle of transparency breaks down.*] Rather than expose the user to locking, locks are automatically assigned at the database file level. [Mat89, Mat87]

SOS

A distributed operating system developed by Shapiro *et. al.* at INRIA. C++ was extended to support a number of features, including object migration and persistence. Their extensions present a simple view of persistent store, providing only a single level of naming, and viewing the process of importing a value from the store as the migration of the object from a server (*i.e.,* one managing the store) to the client. Unlike E and Avalon/C++ there is no support for persistent binding. [SGM89]

ObServer and ENCORE

A Brown project. ObServer is the typeless object-oriented database system and ENCORE the more semantically-oriented system on top of it. ObServer was written general enough to be used by a number of systems requiring database semantics without any associated programming language features (which are supported in ENCORE). In fact, a persistent programming environment, GARDEN, is written on top of ObServer. [Rei86, HZ87]

ORION

A system out of MCC, motivated by interest in building a multimedia development systems. Is distinguished by its extensive support for type evolution, which allows entensive changes to the schema, automatically adjusting the database objects to compensate. Also spurred some work on query optimization. [WK87, KBC+87, KBCG89, KGBW90]

AVANCE

An object-oriented, distributed database programming language. Its most interesting feature was the presence of system-level version control, which is used to support schema evolution, system-level versioning (as a way of improving concurrency), and objects with their own notion of history. System consists of programming language (PAL) and distributed persistent object manager. An ambitious project, AVANCE was unfortunately never completed. [BB88, BH89]

MNEME

A new project to develop a robust object-oriented database system. Current plans exist to use support Modula3 and Smalltalk as base languages, but much of the published work so far deals with the persistent object system. This platform is noteworthy as one of its goals is to be essentially free from the language-specific semantics and to be sympathetic to the needs of the range of design applications. [MS88, Mos89, HMB90, Mos90]

GESTALT

A functional database system. Old versions of objects persist indefinitely, and can always (modulo tape archival) be retrieved. [HN88]

Machiavelli

A new statically-typed persistent language being developed at Penn. [OBBT89]

Perlog

Persistent Prolog. [Mof87]

10.2 Proceedings, Collections, and Surveys

There are a number of conference proceedings, book collections, and surveys that the reader might find an interest in, including:

[AB87] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

An extensive survey about persistence and their type systems. Identifies four typical operations that should be able to be easily performed in a database programming language, and then evaluates various existing database and (early) persistent programming against them. Includes only scant information on object-oriented language efforts.

[ABM85] M. P. Atkinson, O. P. Buneman, and R. Morrison, editors. *Proceedings of the Persistence and Data Type Workshop*, Appin, Scotland, August 1985. Springer-Verlag. *Published as PPRG Persistent Programming Research Report 16, and redone as* [AB88].

The first of four conferences on persistent programming languages and the underlying persistent object managers. Features early papers on PS-Algol, and Galileo, among other languages and systems.

[AB88] Malcolm P. Atkinson and O. Peter Buneman. *Data Types and Persistence.* Topics in Information Systems. Springer-Verlag, 1988. *Based on the proceedings of the First Appin Workshop, which appeared as PPRG Persistent Programming Research Report 16.*

[CC87] Ray Carrick and Richard Cooper, editors. *A Workshop on Persistent Object Systems: Their Design, Implementation and Use*, North Haugh, St. Andrews KY16 9SS, UK, August 1987. University of St. Andrews, Department of Computational Science. Persistent Programming Research Report 44.

Includes papers on Napier, EXODUS, GemStone, and other persistent languages and object systems.

[Ros89] John Rosenberg, editor. *Proceedings of the Third Workshop on Persistent Object Systems: Their Design, Implementation and Use*, Newcastle, Australia, January 1989. The University of Newcastle.

Includes papers on Napier, and E, as well as some describing new project efforts that I have not mentioned.

[DSZ91] Alan Dearle, Gail M. Shaw, and Stan Zdonik, editors. *Implementing Persisteng Object Bases: Principles and Practice: The Fourth International Workshop on Persistent Object Systems.* Data Management Series. Morgan Kaufmann, San Mateo, CA, 1991.

[SW87] Bruce D. Shriver and Peter Wegner. *Research Directions in Object-Oriented Programming.* MIT Press Series in Computer Systems. MIT Press, Cambridge, MA, 1987.

A collection of papers from contemporary research projects concerned with object-oriented languages. One section dealing with persistent and database issues, including [Bee87, SZ87, MS87].

[Dit88] K.R. Dittrich, editor. *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, Ebernburg, West Germany, September 1988. Springer-Verlag.

Produced from the proceedings of the Second International Workshop on Object-Oriented Systems (1988). Includes papers on a host of object-oriented database systems that appear few other places in the literature. Some of the projects are programming languages extended for database applications, such as OPAL, while others are totally redesigned database systems based on object-oriented principles.

[OOP86] *Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, Portland, OR, September 1986.

Includes a track on object-oriented database systems.

[OOP87] *Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, Orlando, FL, September 1987.

Includes a track on object-oriented database systems.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems.* Data Management Series. Morgan Kaufmann, San Mateo, CA, 1990.

A collection of republished papers dealing with the principle and background issues necessary for the building of a object-oriented database programming language. Includes a number of the papers cited in this survey, including [CAC+84, **ABC+83**, **ACO85**, CDG+90, MS87, HZ87] .

[Kim90] Won Kim. *Introduction to Object-Oriented Databases.* Computer Systems. MIT Press, Cambridge, MA, 1990.

A book by the leader of the ORION project, discussing many of the issues involved in the design of a database programming language.

References

[AB87] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[AB88] Malcolm P. Atkinson and O. Peter Buneman. *Data Types and Persistence*. Topics in Information Systems. Springer-Verlag, 1988. *Based on the proceedings of the First Appin Workshop, which appeared as PPRG Persistent Programming Research Report 16.*

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisolm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983. Reprinted in ZdonikMaier90-Readings[ZM90].

[ABM85] M. P. Atkinson, O. P. Buneman, and R. Morrison, editors. *Proceedings of the Persistence and Data Type Workshop*, Appin, Scotland, August 1985. Springer-Verlag. *Published as PPRG Persistent Programming Research Report 16, and redone as* [AB88].

[ACC82] Malcolm Atkinson, Ken Chilsholm, and Paul Cockshott. PS-Algol: An Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[ACC83] M.P. Atkinson, K.J. Chilsholm, and W.P. Cockshott. CMS — a chunk management system. *Software – Practice and Experience*, 13:273–285, 1983.

[ACCM83] M.P. Atkinson, K.J. Chilsholm, W.P. Cockshott, and R.M. Marshall. Algorithms for a persistent heap. *Software – Practice and Experience*, 13:259–271, 1983.

[ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985. Reprinted in ZdonikMaier90-Readings[ZM90].

[ACPP89] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically typed language. Research Report 47, DEC Systems Research Center, Palo Alto, California, June 1989.

[AGOP88] A. Albano, F. Giannotti, R. Orsini, and D Pedreschi. The type system of Galileo. In Atkinson and Buneman [AB88], chapter 8, pages 101–199. *Based on the proceedings of the First Appin Workshop, which appeared as PPRG Persistent Programming Research Report 16.*

[AM84] Malcolm P. Atkinson and Ronald Morrison. Persistent first class procedures are enough. In *Proceedings of the Fourth Conference of Software Technology and Theoretical Computer Science*, volume 181 of *Lecture Notes in Computer Science*, pages 223–240. Springer-Verlag, Berlin, December 1984.

[AM85] Malcolm P. Atkinson and Ronald Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, October 1985.

[AM87] Malcolm P. Atkinson and Ronald Morrison. Polymorphic names, types, constancy and magic in a type secure persistent object store. In Carrick and Cooper [CC87], pages 1–12. Persistent Programming Research Report 44.

[AU79] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 110–120, New York, 1979. ACM.

[BB88] Anders Björnerstedt and Stefan Britts. AVANCE: An object management system. In *Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 206–221, San Diego, CA, September 1988.

[Bee87] David Beech. Groundwork for an object database model. In Shriver and Wegner [SW87], pages 317–354.

[BH89] Anders Björnerstedt and Christer Hultén. Version control in an object-oriented architecture. In Kim and Lochovsky [KL89], chapter 18.

[Blo83] Toby Bloom. Dynamic module replacement in a distributed programming system. Technical Report MIT/LCS/TR-303, Massachusetts Institute of Technology, Cambridge, MA, March 1983. Thesis (Ph.D.).

[BMO+89] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The GemStone data management system. In Kim and Lochovsky [KL89], chapter 12.

[BZ81] M.L. Brodie and S.N. Zilles, editors. *Proceedings Workshop on Data Abstraction, Data Bases, and Conceptual Modelling*, 1981. ACM SIGMOD Special Issue 11,2.

[CAC+84] W.P. Cockshott, M.P. Atkinson, K.J. Chilsholm, P.J. Bailey, and Morrison R. Persistent objected management system. *Software – Practice and Experience*, 14:49–71, 1984. Reprinted in ZdonikMaier90-Readings[ZM90].

[CADA87] R.L. Cooper, M.P. Atkinson, A. Dearle, and D. Abderrahmane. Constructing database systems in a persistent environment. In Pirotte and Vassiliov [PV87], pages 117–125.

[Car86a] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 48–70. Springer-Verlag, Berlin, 1986.

[Car86b] Luca Cardelli. The Amber machine. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, Berlin, 1986.

[CC87] Ray Carrick and Richard Cooper, editors. *A Workshop on Persistent Object Systems: Their Design, Implementation and Use*, North Haugh, St. Andrews KY16 9SS, UK, August 1987. University of St. Andrews, Department of Computational Science. Persistent Programming Research Report 44.

[CDG+90] Michael J. Carey, David J. DeWitt, Goetz Graefe, David M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Skekita, and Scott L. Vandenberg. The EXODUS extensible DBMS project: An overview. In Zdonik and Maier [ZM90]. *Also available as WISC-CS-TR 808.*

[CDRS86] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Skekita. Object and file management in the EXODUS extensible database system. In Yahiko Kambayash, editor, *Proceedings of the Twelveth International Conference on Very Large Data Bases*, pages 91–100, Kyoto, August 1986. Very Large Data Base Endowment.

[CDV88] Michael J. Carey, David J. DeWitt, and Scott L. Vandenberg. A data model and query language for EXODUS. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 413–423, Chicago, IL, June 1988. *Also available as WISC-CS-TR 734.*

[Cla89] Stewart M. Clamen. Towards Avalon/Common Lisp: Remote Lisp evaluation. Avalon Note 14, Carnegie Mellon University School of Computer Science, 1989.

[CLNW90a] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing. Reliable distributed computing with Avalon/Common Lisp. In *Proceedings of the International Conference on Computer Languages*, New Orleans, LA, March 1990. Institute of Electrical and Electronic Engineers Computer Society. *Also available as Carnegie Mellon School of Computer Science Tech Report # CMU-CS-89-186; also an extended abstract appears as "An overview of Avalon/Common Lisp," in the Proceedings of the Third Workshop on Large Grained Parallel Programming (Pittsburgh, PA, October 10-11, 1989).*

[CLNW90b] Stewart M. Clamen, Linda D. Leibengood, Scott N. Nettles, and Jeannette M. Wing. Assessment of the Avalon/Common Lisp implementation. Avalon Note 16, Carnegie Mellon University School of Computer Science, 1990.

[CLNW90c] Stewart M. Clamen, Linda D. Leibengood, Scott N. Nettles, and Jeannette M. Wing. A programmer's guide to Avalon/Common Lisp. Avalon Note 15, Carnegie Mellon University School of Computer Science, 1990.

[CM88] Luca Cardelli and David MacQueen. Persistence and type abstraction. In Atkinson and Buneman [AB88], chapter 3, pages 31–42. *Based on the proceedings of the First Appin Workshop, which appeared as PPRG Persistent Programming Research Report 16.*

[Coc83] William Paul Cockshott. *Orthogonal Persistence.* PhD thesis, University of Edinburgh, February 1983. *Published as Edinburgh TR CST-21-83.*

[Col82] R. Cole, A. J.; Morrison. *An Introduction to Programming with S-algol.* Cambridge University Press, 1982.

[Dat81] C.J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, Reading, MA, 4th edition, 1981.

[Dat87] C.J. Date. *A Guide to Ingres.* Addison-Wesley, Reading, MA, 1987.

[DB88] A. Dearle and A.L. Brown. Safe browsing in a strongly typed persistent language. *Computer Journal*, 31(6):540–544, December 1988.

[Dea89] Alan Dearle. Environments: A flexible binding mechanism to support system evolution. In Shriver [Shr89], pages 46–55.

[DHW88] David Detlefs, Maurice P. Herlihy, and Jeannette M. Wing. Inheritance of Synchronization/Recovery Properties in Avalon/C++. *Computer*, 21(12), December 1988.

[Dit88] K.R. Dittrich, editor. *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, Ebernburg, West Germany, September 1988. Springer-Verlag.

[DSZ91] Alan Dearle, Gail M. Shaw, and Stan Zdonik, editors. *Implementing Persisteng Object Bases: Principles and Practice: The Fourth International Workshop on Persistent Object Systems.* Data Management Series. Morgan Kaufmann, San Mateo, CA, 1991.

[Ghe85] G. Ghelli. La gestione della persistenza dei valori nel linguaggio Galileo. *Scienze dell'Informazione*, 1985.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Har90] Robin Milner; Mads Tofte; Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[HMB90] Antony L. Hosking, J. Eliot B. Moss, and Cynthia Bliss. Design of an object faulting persistent smalltalk. *Submitted to OOPSLA/ECOOP 90*, 1990.

[HN88] Michael L. Heytens and Rishiyur S. Nikhil. GESTALT: An expressive database programming system. *SIGMOD Record*, 18(1):54–67, March 1988.

[HSS89] William H. Harrison, John J. Shilling, and Peter F. Sweeney. Good news, bad news: Experience building a software development environment using the object-oriented paradigm. In OOPSLA89 [OOP89], pages 85–94.

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*. IEEE, July 1987.

[HZ87] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):70–95, January 1987. Reprinted in ZdonikMaier90-Readings[ZM90].

[KBC+87] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Composite object support in an object-oriented database system. In OOPSLA87 [OOP87], pages 118–125.

[KBCG89] Won Kim, Nat Ballou, Hong-Tai Chou, and Darrell Garza, Jorge F. Woelk. Features of the ORION object-oriented database system. In Kim and Lochovsky [KL89], chapter 11.

[KGBW90] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the orion next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–24, March 1990.

[Kim90] Won Kim. *Introduction to Object-Oriented Databases*. Computer Systems. MIT Press, Cambridge, MA, 1990.

[KL89] Won. Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, Reading, MA, 1989.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[LDH+87] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. Argus Reference Manual. Technical Report TR-400, MIT Laboratory for Computer Science, Cambridge, MA, November 1987.

[Mat85] David C. J. Matthews. Poly manual. *ACM SIGPLAN Notices*, 20(9):52–76, September 1985. *Also available as University of Cambridge Computer Lab TR #63.*

[Mat86] David C.J. Matthews. An overview of the poly programming language. Technical Report 99, University of Cambridge Computer Laboratory, Cambridge, UK, November 1986.

[Mat87] David C.J. Mathews. A persistent storage system for Poly and ML. Technical Report 102, University of Cambridge, Cambridge, UK, January 1987. *Also published as part of* [Mat89].

[Mat89] David C.J. Matthews. Papers on poly/ml. Technical Report 161, University of Cambridge Computer Laboratory, Cambridge, UK, February 1989. *A collection of previously written papers and research documents on Poly and Poly/ML, some of which are available individually.*

[MBC+88] R. Morrison, A.L. Brown, R. Carrick, R. Conner, and A. Dearle. On the integration of object-oriented and process-oriented computation in persistent environments. In Dittrich [Dit88], pages 334–339.

[MBC+89a] R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle, and M.P. Atkinson. The napier type system. In Rosenberg [Ros89], pages 253–269.

[MBC+89b] R. Morrison, A.L. Brown, R. Carrick, R. Connor, A. Dearle, M.J. Livesey, C.J. Barter, and A.J. Hurst. Language-design issues in supporting process-oriented computation in persistent environments. In Shriver [Shr89], pages 736–744.

[MBW80] J. Mylopoulos, P.A. Bernstein, and H.K.T. Wong. A language facility for designing database intensive applications. *ACM Transactions on Database Systems*, 5(2):185–207, June 1980. Reprinted in ZdonikMaier90-Readings[ZM90].

[Mis84] Nathaniel Mishkin. Managing persistent objects. Research Report 338, Yale University, New Haven, Conn., 1984. *Adapted from PhD Thesis.*

[MK84] D. McCleod and R. King. Semantics database models. In S.B. Yao, editor, *Principles of Database Design*. Prentice-Hall, 1984.

[Mof87] D. Moffat. Modular requirements in persistent prolog. In Carrick and Cooper [CC87], pages 68–77. Persistent Programming Research Report 44.

[Mor79] R. Morrison. S-algol language reference manual. Technical Report CS/79/1, University of St. Andrews, 1979.

[Mos89] B. Moss, J. Eliot. The Mneme peristence object store. COINS TECHNICAL REPORT 89-107, University of Massachusetts, Amherst, MA, October 1989.

[Mos90] B. Moss, J. Eliot. Working with persistent objects: To swizzle or not to swizzle. COINS TECHNICAL REPORT 90-38, University of Massachusetts, Amherst, MA, May 1990.

[MS86] David Maier and Jacob Stein. Indexing in an object-oriented DBMS. In Klaus R. Dittrich and Umeshwar. Dayal, editors, *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, pages 444–452, Pacific Grove, CA, September 1986. Institute of Electrical and Electronic Engineers.

[MS87] David Maier and Jacob Stein. Development and implementation of an object-oriented DBMS. In Shriver and Wegner [SW87], pages 355–392.

[MS88] J. Eliot B. Moss and Steven Sinofsky. Managing persistent data with Mneme: Designing a reliable, shared object interface. In Dittrich [Dit88], pages 298–316.

[MSOP86] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. In OOPSLA86 [OOP86], pages 472–482.

[OBBT89] Atsusi Ohori, Peter Buneman, and Val Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 46–57, Portland, OR, May 1989.

[OOP86] *Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, Portland, OR, September 1986.

[OOP87] *Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, Orlando, FL, September 1987.

[OOP89] *Proceedings of the ACM Conference on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, New Orleans, LA, October 1989.

[Phi87] P. Philbrow. Associative storage and retrieval: Some language design issues. In Carrick and Cooper [CC87], pages 226–232. Persistent Programming Research Report 44.

[PS87] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In OOPSLA87 [OOP87], pages 111–117.

[PSM87a] D. J. Penney, J. Stein, and D. Maier. Is the disk half full or half empty? *combining optimistic and pessimistic concurrency mechanisms in a shared persistent object base.* In Carrick and Cooper [CC87], pages 337–345. Persistent Programming Research Report 44.

[PSM87b] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an object server with other worlds. *ACM Transactions on Office Information Systems*, 5(1):27–47, January 1987.

[PV87] A. Pirotte and T. Vassiliov, editors. *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, Brighton, 1987. Very Large Data Base Endowment.

[RC87] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In U. Dayal and I. Traiger, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 208–219, San Francisco, CA, May 1987.

[RC89a] Joel E. Richardson and Michael J. Carey. Implementing persistence in e. In Rosenberg [Ros89], pages 302–319. *Republished as part of* [**RC89b**].

[RC89b] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software – Practice and Experience*, 19(12):1115–1150, December 1989.

[RCDS87] J.E. Richardson, M.J. Carey, Dewitt D.J., and D.T. Schuh. Persistence in EXODUS. In Carrick and Cooper [CC87], pages 96–113. Persistent Programming Research Report 44.

[Rei86] S.P. Reiss. An object-oriented framework for graphical programming. *SIGPLAN Notices*, 21(10), October 1986. *also available as BROWN-TR CS-86-17.*

[Ros89] John Rosenberg, editor. *Proceedings of the Third Workshop on Persistent Object Systems: Their Design, Implementation and Use*, Newcastle, Australia, January 1989. The University of Newcastle.

[RWW89] William R. Rosenblatt, Jack C. Wileden, and Alexander L. Wolf. Oros: Toward a type model for software development environments. In OOPSLA89 [OOP89], pages 297–304.

[SBD+86] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson. The Camelot Project. *Database Engineering*, 9(4), December 1986. Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.

[Sch77] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977.

[SGM89] Marc Shapiro, Phillippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In *Proccedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Nottingham, UK, July 1989. Springer-Verlag.

[Shr89] B.D. Shriver, editor. *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences.*, volume 2: Software Track, Kailua-Kona, HI, January 1989. Institute of Electrical and Electronic Engineers, IEEE Computer Society Press.

[SM83] J.W. Schmidt and M. Mall. Abstraction mechanisms for database programming. *SIGPLAN Notices*, 18(6), June 1983.

[Ste90] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, second edition edition, 1990.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[SW87] Bruce D. Shriver and Peter Wegner. *Research Directions in Object-Oriented Programming*. MIT Press Series in Computer Systems. MIT Press, Cambridge, MA, 1987.

[SZ87] Andrea H. Skarra and Stanley B. Zdonik. Type evolution in an object-oriented database. In Shriver and Wegner [SW87], pages 393–415. *An early version of this paper appears in the OOPSLA '86 proceedings.*

[Wai87] F. Wai. Distribution and persistence. In Carrick and Cooper [CC87], pages 207–225. Persistent Programming Research Report 44.

[Wai89] F. Wai. Distributed ps-algol. In Rosenberg [Ros89], pages 343–357.

[WHC+88] Jeannette Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietske, Richard Lerner, and Su Yuen Ling. The Avalon/C++ programming language (version 0). Technical Report CMU-CS-88-209, Carnegie Mellon University School of Computer Science, December 1988.

[WHC+91] Jeannette Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietske, Richard Lerner, and Su Yuen Ling. The Avalon language. In Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors, *Camelot and Avalon: A Distributed Transaction Facility*, The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo, California, February 1991.

[WK87] Darrell Woelk and Won Kim. Multimedia information management in an object-oriented database system. In Pirotte and Vassiliov [PV87], pages 319–329.

[WSK+81] A.I. Wasserman, D.D. Shertz, M.L. Kersten, R.P. Reit, and M.D. van de Dippe. Revised report on the programming language PLAIN. *ACM SIGPLAN Notices*, 1981.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems*. Data Management Series. Morgan Kaufmann, San Mateo, CA, 1990.

A Persistent Language Features

The following appendix is a condensation of the information presented in the body of the paper. Possible approaches to the various research issues are enumerated, followed by a table which categorizes each featured language system accordingly.

1. **Communication Initialization Model**
   (Section 2.1.1)
   (a) Single database, implicit initialization.
   (b) Multiple database, explicit file manipulation.
   (c) Multiple database, with default partition definable.

2. **Communication Model**
   (Section 2.1.1)
   (a) Explicit import/export. Operations to move object from persistent to volatile area.
   (b) Fully transparent scheme. Persistent objects automatically imported from persistent store on demand. Similar to virtual memory systems.
   (c) Mixed scheme. Mostly transparent access to the data in the store with some support for locking/concurrency. Transparency is compromised some, since it is possible that a request for a persistent value would block.

3. **Persistent Object Identity**
   (Section 2.1.1)
   (a) Yes. Identity is preserved when an object is maintained in the persistent store.
   (b) No. The process of installing a value into the store is not identity-preserving.
   (c) Not applicable. The values that can be maintained in the persistent store are too simple to be able to detect identity (*i.e.,* no pointer references).

4. **Pervasion of Persistence**
   (Section 2.1.2)
   (a) Persistence is universally applied. All objects in the language may persist. (Orthogonal Persistence)
   (b) Persistence is restricted to instances of certain types.
   (c) Persistence is restricted to a fixed set of type instances.

5. **Declaring/Denoting Persistent Objects**
(Section 2.1.3)

   (a) Instances of special types. All instances of special "persistent" types persist, although they might not be addressable. An explicit delete operation is provided to remove objects from the store.

   (b) Based on GC reachability. Any object reachable from a system-defined persistent root will persist.

   (c) All addressable objects persist. This is a special instance of the previous case, where the persistent root and the garbage-collection root are the same.

6. **Naming of Persistent Objects**
(Section 2.1.4)

   (a) Lexical lookup (*e.g.,* modules, static environments) built into the language.

   (b) Specialized dictionary data types to assist in arranging the data.

   (c) Flat, top-level naming (*e.g.,* files, processes, type names).

   (d) Keyed on top-level names.

7. **Implementation Features**
(Section 2.1.7)

   (a) Page Faulting. Using virtual memory features to reduce cost of detecting persistent addresses.

   (b) Clustering. System accepts hints from client regarding reference patterns, and tries to reduce working set by collecting objects.

   (c) Swizzling. Persistent addresses replaced by local ones when object is interned. Pays off when object is referenced often while it is in memory.

   (d) Organizing data for optimizing queries (à la relational database systems.)

   (e) Index tables. Associative access supported on elements for efficient lookup.

8. **Database-oriented Types and Control Forms**
(Section 2.2.2)

   (a) Aggregates (related to addressing). Collections of like-typed objects with functions and forms to operate on or search over all of them.

   (b) Index tables. Merely a language-level view of the implementation feature noted above.

   (c) Classification. Using the type hierarchy to help organize data.

9. **Type Evolution**
   (Section 2.1.6)
   (a) Type identity by name; type can be redefined, making old versions inaccessible (or worse). [Dynamic Typing]
   (b) New type definition aliases previous one; but old instances (and old type definition) still around. [Static Typing]
   (c) Direct Support for Type Evolution: Emulation
   (d) Direct Support for Type Evolution: Eager Conversion
   (e) Direct Support for Type Evolution: Lazy Conversion

10. **Locking Granularity**
    (Section 2.2.4)
    (a) On database files
    (b) On pre-defined regions of files
    (c) On objects
    (d) On parts of objects
    (0) None at all
    (-) Locking is automatic.
    (+) Some programmer control over locking is supported.

11. **Transactions**
    (Section 2.2.3)
    (a) One-per-program
    (b) Transaction blocks (simple transactions only)
    (c) Transaction blocks (simple and nested transactions)
    (0) None
    (-) Transaction management is automatic.
    (+) Programmer control over transaction management is supported.

12. **Distribution**
    (Section 2.3)
    - Does the system support distributed computation?

13. **User-level Security**
    (Section 2.3)
    - Does the language system provide a method for enforcing personal-level security on the objects in the persistent store?

System/Language	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
Pascal/R	b	c	n.a.	c	a	d	e	ab	a	a−	a−	no	no
PS-Algol	b	c	yes	a	b	b	bc	a	b	a−	a−	no	no
Napier	b	c	yes	a	b	a	bc		b	a−	a+	no	no
Amber	b	a	no	a	a	c			a	0	0	no	no
Galileo	a	b	yes	a	c	da	ce	abc	b	0	c+	no	no
Exodus/E	a	c	yes	b	a	da	b	a	a	cd+	b−	no	no
GemStone/OPAL	a	c	yes	a	c	b	acde	abc	bc[1]	c+?	b+	yes	yes
Avalon/C++	b	c	yes[2]	b	a	c	a		a	cd+	c+	yes	no
Avalon/CL	a	a	yes	b	a	a			a	c−	c+	yes	no
INGRES/SQL	a	c	n.a.	c	a	d	de	ab	d	c−	b+	yes	yes
AVANCE	a?	c	yes	b	a	?		a[3]	c	d−?	c+	yes	no

n.a.=not applicable

?=insufficient information in literature

[1]Eager (in current implementation); lazy and eager at GC (ultimately).

[2]Identity is only preserved on the resident server. Objects passed out of a server through the RPC mechanism are copies.

[3]Arbitrary size restriction on size of aggregates.

B Relational Database Systems

Much of the database system effort of the past fifteen years has been in the area of relational database systems, so I will include here a short primer on relational database systems from the perspective of someone familiar with traditional programming languages. [*Material gleaned from* [Dat81].]

The basic unit of information in relational systems is the **record**, often called a **tuple** or **row**. Records are tuples of typed, named **fields** — also called **attributes** or **columns** — consisting of primitive values (strings, integers, reals, dates) only.

Records are stored according to type in unordered collections called **tables**, which are sometimes referred to as **relations** or **files**. From a programming language perspective, one might view a table as a datatype, and a record as an instance. Figure 1 (p.64), provides some table definitions, along with some sample data records.

Operations on tables generally take the form of queries, retrieving data out of existing tables and returning new ones. (Sample queries can be seen in Figure 2 (p.65).) There are also operations to add and delete records from a table, and to modify the value of a record field. A fixed set of **aggregate operators** — COUNT, SUM, AVG, MIN, and MAX — are provided.

A little bit of nomenclature: a **candidate key** on a relation is a set of attributes that uniquely identify the records of a table — that is, it can never be the case that a relation will have two tuples with the same values for those attributes. In our example, the airline (AL), flight number (NO), and destination (ARR) attributes are a candidate key for the `FLIGHTS` relation. AL, NO, and origin (DEP) is another. From the set of candidate keys, one is arbitrarily chosen as the **primary key**, and is used by the system as an associative address for the tuple. Database programmers are encouraged to declare a primary key, so that the database management system (DBMS) can optimize table operations.

Relational databases have no pointers: references between records are achieved via a correspondence between sets of fields. In the second query example in Figure 2 (p.65), the AL, DEP, and ARR columns in the `FLIGHTS` table are used to uniquely identify a row in the `PRICES` table. These attributes constitute a **foreign key** in the `FLIGHTS` table for the `PRICES` relation. Formally, a foreign key is an attribute set from one relation that is required to match the primary key of some (not necessarily different) relation.

In a database with large amounts of records, retrieval speed is of primary importance. To this end, relational database systems support **index** tables. Keyed upon by a unique subset of attribute values, usually the primary key[1], an index provides a method of retrieval faster than sequential search, at the expense of some additional bookkeeping requirements whenever the table or its records are modified. Usually, the database manager must explicitly create index tables, specifying the index fields.

[1] An index on a primary key is called a **primary index**.

```
CREATE TABLE FLIGHTS
(   AL      CHAR(2)
    NO      SMALLINT ;
    DEP     CHAR(3) ;
    ARR     CHAR(3) ;
    DTIME   DATE ;
    ATIME   DATE  ) ;
```

```
CREATE TABLE PRICES
(   AL     CHAR(2)
    DEP    CHAR(3) ;
    ARR    CHAR(3) ;
    COST   MONEY ) ;
```

AL	NO	DEP	ARR	DTIME	ATIME
CP	918	YYZ	PIT	10:55am	12:00pm
CP	917	PIT	YYZ	12:23pm	1:23pm
NW	1481	DET	PIT	6:25am	7:25am
NW	1418	PIT	DET	7:45am	8:45am
NW	1421	DET	PIT	1:25pm	2:25pm
NW	1414	PIT	DET	2:45pm	3:45pm
US	709	YUL	SYR	11:30am	12:25pm
US	709	SYR	PIT	12:55pm	2:00pm
US	554	PIT	SYR	1:15pm	2:15pm
US	554	SYR	YUL	2:50pm	3:38pm
US	172	CLE	PIT	7:10am	7:51am
US	317	CLE	PIT	8:20am	9:07am
US	79	PIT	CLE	8:45am	9:29am
US	263	PIT	CLE	12:00pm	12:43pm
US	1769	PIT	SYR	8:23pm	9:10pm

AL	DEP	ARR	COST
CP	PIT	YYZ	$165.50
NW	DET	PIT	$175.00
US	CLE	PIT	$145.50
US	PIT	YUL	$201.50
US	PIT	SYR	$159.50

Figure 1: Definitions of two tables in the SQL database definition language, and associated records.

```
SELECT  AL, NO
FROM    FLIGHTS
WHERE   ARR = 'PIT'
AND     ATIME < '12pm'
AND     ATIME > '6am'
```

AL	NO	DTIME
NW	1481	7:25am
US	172	7:51am
US	317	9:07am

```
SELECT  FLIGHTS.*, PRICES.COST
FROM    FLIGHTS, PRICES
WHERE   FLIGHTS.AL = PRICES.AL
AND     (  FLIGHTS.ARR = PRICES.DEP
        OR FLIGHTS.ARR = PRICES.ARR )
AND     (  FLIGHTS.DEP = PRICES.DEP
        OR FLIGHTS.DEP = PRICES.ARR )
AND     PRICES.COST =
        (    SELECT  MIN (COST)
             FROM    PRICES ) ;
```

AL	NO	DEP	ARR	PRICE
US	331	PIT	CLE	$145.50
US	332	CLE	PIT	$145.50
US	333	PIT	CLE	$145.50
US	334	CLE	PIT	$145.50

Figure 2: Two sample queries in the flight database: The first asks for all morning flights arriving in Pittsburgh, while the second asks for the cheapest flight(s) leaving Pittsburgh. (This operation, performing a cross-reference between tables, is called a **join**.)

Transactions and Fault-Tolerance in Relational Database Systems

With the records comprising the database persisting indefinitely, it is important that some provision is made to protect the data against (hardware or software) faults. Assuming for the time being that the secondary storage medium (the disk) is trustworthy, we only have to worry about the computer crashing while the database system is in the midst of performing an update. The abstraction typically used to deal with this problem is the **transaction**, which guarantees that the operations associated with it will be performed *atomically* — either all the operations within the transaction are performed or none. When the subprogram run as a transaction completes, its effects are installed, and the transaction is said to have **committed.** However, if the subprogram is preventing from committing — the machine might have crashed while it was in midst of the computation, for instance — the effects are not installed, and the transaction is said to have **aborted.** Transactions are a powerful mechanism for guaranteeing that the database is always left in a consistent state.

There are a number of ways a system can protect itself against data corruption on the disk. The most popular methods are **replication** and **backup storage**. In brief, replication involves duplicating the database onto other disks, so data corrupted in one place can still be retrieved on others. This procedure increases the expense of writing to the database, but increases the reliability. Backup involves archiving the database onto slower, yet more reliable media, such as tape, from which it can be restored in an emergency.

Index

Below is an index of the primary sources of the language systems featured in this survey: