VLDB Journal, 4, 319-401 (1995), Malcolm Atkinson, Editor
©VLDB

Orthogonally Persistent Object Systems

Malcolm Atkinson and Ronald Morrison

Received June, 1993; revised version received, March, 1995; accepted April, 1995.

Abstract. Persistent Application Systems (PASs) are of increasing social and economic importance. They have the potential to be long-lived, concurrently accessed, and consist of large bodies of data and programs. Typical examples of PASs are CAD/CAM systems, office automation, CASE tools, software engineering environments, and patient-care support systems in hospitals. Orthogonally persistent object systems are intended to provide improved support for the design, construction, maintenance, and operation of PASs. Persistence abstraction allows the creation and manipulation of data in a manner that is independent of its lifetime, thereby integrating the database view of information with the programming language view. This yields a number of advantages in terms of orthogonal design and programmer productivity which are beneficial for PASs. Design principles have been proposed for persistent systems. By following these principles, languages that provide persistence as a basic abstraction have been developed. In this paper, the motivation for orthogonal persistence is reviewed along with the above mentioned design principles. The concepts for integrating programming languages and databases through the persistence abstraction, and their benefits, are given. The technology to support persistence, the achievements, and future directions of persistence research are then discussed.

Key Words. Orthogonal persistence, persistent programming languages, database programming languages, persistent application systems.

1. Introduction

This paper presents a broad review of current research and achievements in orthogonally persistent object systems. It provides an advanced tutorial for those commencing research or advanced study in persistence, databases, or database programming languages, and a survey of the persistent language community's con-

Malcolm Atkinson, Ph.D., is Professor, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, *mpa@dcs.glasgow.ac.uk*, and Ronald Morrison, Ph.D., is Professor, School of Mathematical and Computational Sciences, University of St. Andrews, North Haugh, St. Andrews KY16 9SS, Scotland, *ron@dcs.st-andrews.ac.uk*.

tribution to this growing research area. A particular goal is to summarize the existing achievements and to identify the current research issues.

Orthogonally persistent object systems support a uniform treatment of objects irrespective of their types by allowing values of all types to have whatever longevity is required. The motive for establishing this uniform treatment is presented first.

Subsequent sections of this paper present definitions, integration concepts, technology to support persistence and achievements. Table 1 identifies the major topics covered. Section 7 suggests some of the crucial areas of further research.

1.1 Incoherence of Current Technology

The aim of persistent programming is to support the activity of applications' construction for long-lived, concurrently accessed, and potentially large bodies of data and programs; referred to here as Persistent Applications Systems (PASs). They are designated persistent application systems because the application often outlives its individual components, and even its implementation technology. Typical examples of such PASs are: CAD/CAM systems, office automation, CASE tools, software engineering environments (Teitelbaum and Reps, 1981; Akima and Ooi, 1989; Bott, 1989; Sommerville et al., 1989; Thomas, 1989), integrated hospital administration and medical systems, large scientific databases and programs that analyze them, geographic information systems, environmental modeling systems, object-oriented databases (Bancilhon et al., 1988; Bretl et al., 1989), and process modeling systems (Bruynooghe et al., 1991; Curtis et al., 1992; Han and Welsh, 1993).

Those that commission the construction of a PAS expect it to be built at reasonable cost, to be delivered on time, and to serve their organization reliably for many years, accommodating change as requirements change and as technology advances. They are often disappointed since it frequently proves much harder and more expensive to build and maintain a PAS than was expected and its evolution is invariably problematic.

This may be illustrated with an example. In a health care management system in use in a hospital, the medical staff were alarmed when many of the fields in the medical records disappeared. Their local support staff diagnosed a fault in the database and restored it from an earlier state. The problem *persisted*, so they made an earlier restoration, losing more information. Still the problem persisted. The suppliers of the PAS were called in and, eventually, they discovered that a file containing a font used by the user interface management system had been lost. Inconsistent behavior by the supporting subsystems had led to a very expensive service interruption and the loss of information. If restoration had functioned identically for all the data, then the lost file would have been restored. As different data were treated differently, the local staff found the system incomprehensible, and the implementors had a struggle to understand the cause of the failure. Similar, but sometimes more detailed, inconsistencies bedevil every stage of PAS construction and maintenance when it is constructed using traditional technology.

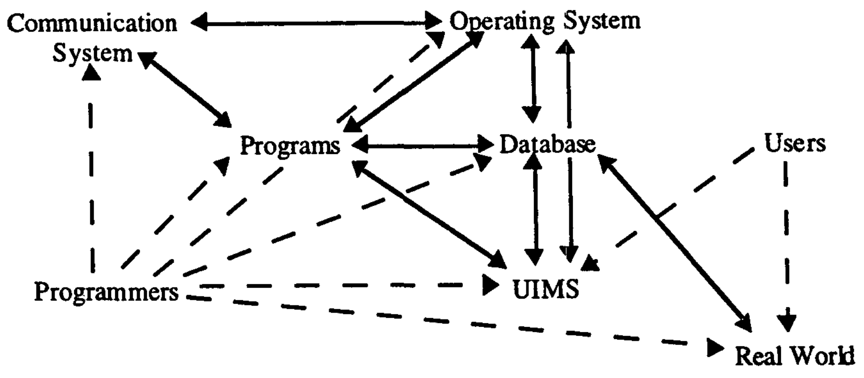
Table 1. Location of major topics in this paper

Topic	Section
Problems with Conventional Technology	1.1, 1.2
Principles of Persistence & Definition of Orthogonal Persistence	2
Benefits of Orthogonal Persistence	2.3
Integration of Databases & Programming Languages	3
Types & Data Models	3.1
Persistent Type Systems	3.1.1
Binding Mechanisms	3.2
Concurrency and Transactions	3.3
Technology to Support Orthogonal Persistence	4
Implementation Architectures	4.1
Implementing Persistence by Reachability	4.2
Type-Safe Linguistic Reflection	4.3
Incremental Persistent Application System Construction	5.1
Hyper-Programming	5.2
Persistent Software Engineering	5.3, 5.4
Research Directions	6
Extensions of Persistence	6.1
Exploitation of Persistence	6.2
Delivering Persistence	6.3
Summary & Conclusions	7

Currently, the technology underlying PAS building (called "the support system" in this paper) relies on a number of disparate mechanisms and philosophical assumptions for support and efficient implementation (Atkinson et al., 1983*a*). Among these are operating systems, communications systems (often bundled with the operating system but designed separately), database systems, user interface systems, command languages, editors, file systems, compilers, interpreters, linkage editors, binders, debuggers, DBMS—DDLs and DMLs, query languages, user interface management systems (UIMS), transaction managers, concurrency managers, and machine types. Programmers have to cope with variations such as different naming, type, and binding schemes in each of the components they use. Perhaps the hardest challenge these programmers face is coping with different recovery, concurrency, and transactional behavior.

The incoherence and complexity arising from utilizing these many different and diverse application and system building mechanisms increases the cost both

Figure 1. Complex inter-dependence in current systems



intellectually and mechanically of building even the simplest of systems. The complexity distracts the application builders from the task in hand, forcing them to concentrate on mastering the multiplicity of programming systems rather than the application being developed. Perversely, the plethora of disparate mechanisms is also costly in machine terms in that the code for interfacing them, their redundant duplication of facilities, and their contention for resources cause execution overheads.

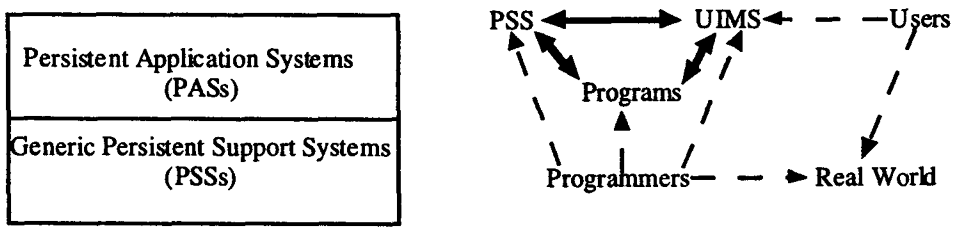
Figure 1 illustrates the interconnection of system components and users of the system. In moments of crisis and during peaks of activity, we expect the most from our PASs. Unfortunately, it is precisely at such moments of stress that they malfunction, due to the complexity of inter-dependencies between components.

In Figure 1, the solid lines represent data flow, translation, and mappings; the dashed lines what users and programmers are required to understand. The situation shown is ideal for the users; they are required to understand only the minimum. However, observations of real applications show that the users find they have to learn about aspects of the operating system and database. On the other hand, the situation is far from ideal for application programmers, as they have to deal with *four* subsystems as well as the programs and the applications that are their real concern.

Individual inconsistencies between different underlying subsystems present software engineers working on a PAS with enough problems. However, each pair of subsystems presents these discontinuities, and the composition of complexity is often multiplicative rather than additive. Recognition of the cost of such disharmonies has motivated the designers of orthogonally persistent object systems to propose and support computational models that allow large scale and long-lived computation without these unnecessary sources of complexity.

In many cases, the inconsistent treatments are not fundamental but accidental (Atkinson, 1978). The various subsystems were built at different times, when the engineering trade-offs were different. In consequence, they provide virtually the

Figure 2. Target architecture to support persistent application systems


Persistent Application Systems (PASs)
Generic Persistent Support Systems (PSSs)
PSS
UIMS
Users
Programs
Programmers
Real World

Proposed Architecture

Simplified Context for Programming and Use

same services, but inconsistently, as they were designed and developed independently. By choosing to provide the total composition of services needed to support a PAS within one coherent design, the designers of orthogonally persistent object systems have eliminated the accidental disharmonies, and exposed the few places where it is a difficult research issue to find the integrating concepts.

It is now timely to propose new models of computation, expressed both linguistically and architecturally, in response to the advent of new hardware technologies with different time, space, and cost trade-offs, and to the development of new application areas, such as office automation, CAD/CAM, and CASE. Persistent systems are one such proposal.

In summary, it is important to distinguish carefully between a PAS and the support system that enables it to operate, and application programmers who build and maintain it. As the cost of building the support system can be amortized over many PASs, and as the cost of application programming dominates the software industry, it is appropriate to invest in improved support systems to reduce PAS programming costs. Orthogonal persistence provides the design principles to guide this investment. The desired simple relationship between a PAS and a generic support system is shown in Figure 2.

1.2 Analysis of the Causes of Incoherence

Some of the complexity in a PAS is intrinsic in that it emanates from the tasks that the PAS sets out to support. However, as argued above, a considerable part of this complexity is extraneous, an unfortunate artifact of the independent development of several essential supporting technologies. Before proposing integrated design principles that eliminate this extraneous complexity, its origins are analyzed.

This analysis focuses on the relationship between programming languages and database systems for three reasons:

- it is typical of the class of problems that arises (e.g., comparing operating systems and programming languages would reveal similar extraneous differences);
- it is the domain that has received most attention; and
- it matches the interests of the expected audience of this paper.

The database and programming language communities have continued to research and develop products independently of one another, despite having to provide many similar services. For example, each provides the means of naming values, each allocates space in which to store values, each provides a means of constraining those values and the operations on them, each provides modeling and description mechanisms, each provides mechanisms for extracting values from data structures, and each provides concurrency.

The inconsistencies originate from different philosophies and circumstances in the two camps. The database community was faced with severe engineering problems. It is difficult to preserve large volumes of data reliably, and to support many processes operating against those data efficiently. It is also difficult to accommodate the design habits of the system analyst, and to allow virtually independent teams of programmers to develop parts of a PAS independently. Consequently, technical solutions dominate. Relations, query optimization, serializable transactions, and views are examples of their success.

By contrast, programming language designers have typically aspired to help programmers be precise, and to make programs understandable. They have sought design principles that lead to languages with regular rules. These rules should match programmers' intuitions, be easy to define precisely, and lead to programs that can be executed with reasonable efficiency.

Of course, there are exceptions in each camp. Relations have simple, formal, and precise rules, and there is an unfortunate number of programming languages which do not. The responses to design difficulties in each context is informative. In programming languages, essential requirements are often ignored; for example, many languages have had no predefined treatment of I/O, and functional languages offer no mechanism for update. In databases, a solution is often designed independently and simply added (e.g., long-running transactions). These additions may not compose well with the existing constructs.

The separate development and consequent inconsistencies tend to perpetuate and grow. The intellectual and software investment in each camp militates against easy adoption of the other's ideas. The dichotomy between philosophies continues and may be heightened as they view each other through caricatures. To a programming language designer, the database world looks like a mess of incomprehensible ad hoc design with little underlying philosophy. On the other hand, database designers are surprised that programming languages are so unhelpful with real problems such as bulk types, persistence, concurrency, and transactions.

Table 2. Various life-times of data values

1	Transient results in expression evaluation
2	Local variables
3	Global variables and heap items
4	Data that lasts a whole execution of a program
5	Data that lasts for several executions of several programs
6	Data that lasts for as long as a program is being used
7	Data that outlives a succession of versions of such a program
8	Data that outlives versions of the persistent support system

The challenge is to show that they need not remain behind separate barricades, and that PAS builders can benefit from the resultant integrated support environment, rather than suffer the task of bridging a perpetuated and possibly widening gulf.

1.3 Introduction to Persistence

Before introducing persistence, it may be important to point out one approach to the above incoherence that does not lead to a solution. That approach is to perpetuate the current underlying technologies, by gluing them together with enough glue-ware, and hiding them behind sufficient "standard" interface veneers. The underlying differences in semantics ultimately show through as, for example, in failure semantics when the combined system is stressed. Since PAS builders wish to support nearly continuous availability, or at least explain to users what is happening when service commitments are not met, they have to understand the underlying technology, including *all* of its different behaviors. Thus, they are still concerned with the very complexity from which we seek to free them.

The approach that appears to minimize disharmony most effectively is that taken in the design and provision of *orthogonally persistent object systems*, and particularly *persistent programming languages*. In this approach, well-tried language design principles are adhered to, but they are applied to deliver a complete computational environment.

The term *persistence* is used variously in common parlance, but has been defined in this context to mean supporting data values for their full life times however brief or long these may be. The life times of data values are the period from their creation until they are no longer used by the persistent application. This range is illustrated in Table 2.

Typically, rows 1 to 4 have been serviced by programming languages, and rows 5 to 8 by databases and file stores. Perversely, a barrier also exists between the forms of the data used. Short-term persistent data are often presented to the user as a collection of base types that may be aggregated or composed into constructed forms such as records, variants, functions, and abstract data types. On the other hand, long-term persistent data often take the form of byte streams if held in a file system or some other structure such as relations in databases. Orthogonally persistent systems are designed so that the treatment of data values is uniform and independent of their longevity, size, and type. Their goal is to achieve this uniformity for all aspects of the system services, from data definition and operations, to integrity, concurrency, and distribution.

It should be noted that this goal for the provision of orthogonal persistence is independent of the choice of, for example, data model, type system, and concurrency control mechanism that defines which values may occur and which operations may be applied to them. These choices are made for other reasons guided by their appropriateness for the application domains. However, there are some constraints on this choice if a safe system is to be provided for the PAS builder. Also, the engineering challenges in building such systems are such that these choices are not entirely independent of one another. This issue is investigated in later sections of this paper.

Similarly, the choice of programming paradigm (query language, imperative language, logic language, object-oriented language or functional language) is also orthogonal to the issue of providing persistence. Once again, appropriateness for the application will guide the choice.

The simplification achieved by orthogonal persistence is summarized in Figures 3 and 4.

In Figure 3, a fragment of a PAS is represented, built out of a combination of programming language and database facilities. Here the programmers have to build and understand two models of the same external (real world) system that the PAS supports. Furthermore, they have to ensure that the view and behavior of these models remains precisely consistent when accessed directly via database facilities, and when accessed via programs using the database. This maintenance of precise consistency under all circumstances, including stress, is extremely hard for the application programmers to achieve. Either they achieve it with immense effort, or they fail and the end-users suffer a less comprehensible system. There are other costs when construction is based on two subsystems. Programmers have to write translations between the two systems and computers have to execute them. Often this proceeds via the lowest common denominator of base types, losing structural information and protection.

In Figure 4, the same PAS fragment is envisaged, constructed in an orthogonal persistent system. As the data are supported consistently whatever happens, the programmers only need to understand one model, and maintain one mapping. The reduction in complexity for the programmers is obvious. It should also present a more comprehensible PAS behavior to end-users.

Figure 3. Three mappings with two support systems

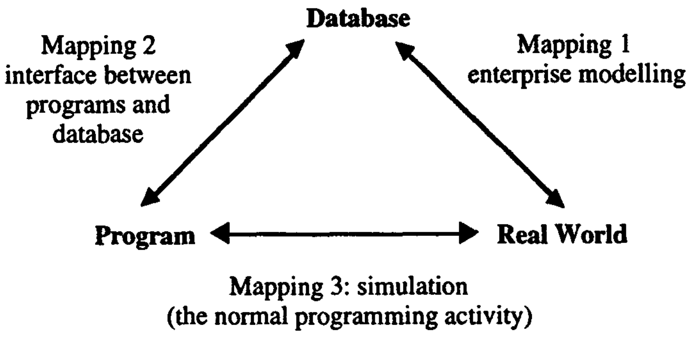
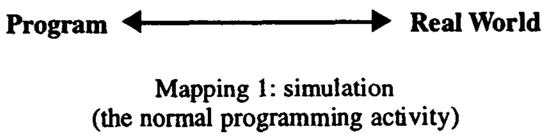


Figure 4. One mapping on an integrated support system



2. Principles of Persistence

To recognize when good design has been accomplished, it is necessary to have some design principles against which to judge the result. In constructing persistent systems, the persistence research community has taken the view that the integrated approach pioneered by language designers should be extended to be more computationally complete. We regard the necessity of using some subsystem that is not specified (or specifiable) in the language as a failure of computational completeness. For example, using a file system is such a failure, since its semantics varies with the context of a program. These principles are developed by extending the design principles that work well for programming languages to encompass the requirements of persistence.

2.1 Design Principles

It has been observed that expressive power in programming languages could be gained by separating the concepts, and allowing them to be combined by powerful composition rules (McCarthy et al., 1962; van Wijngaarden et al., 1969; Strachey, 1967; Tennent, 1977). Strachey (1967) and Tennent (1977) distilled these ideas into three principles for use in the design of programming languages:

1. the principle of correspondence,
2. the principle of abstraction and
3. the principle of data type completeness.

The principle of correspondence states that the rules governing the use of names and bindings in a programming language should be consistent. In particular, the rules for introducing names and bindings in declarations should have a corresponding mechanism for abstraction parameters. This ensures that formal parameters behave consistently with local declarations. The principle of abstraction states that, for all significant syntactic categories in the language, there should be an abstraction mechanism. This allows essential details to be ignored by concentrating on the general structure. An abstraction consists of naming the syntactic category and allowing it to be parameterized. The most widely used forms of abstractions are functions and abstract data types. The principle of data type completeness states that any combination or construction of data should be allowed in all types. As a consequence, all data objects in a language should have the same "civil rights."

The overall goal of the above principles is to design languages that are both simple and powerful. They are simple in that there is a minimum of defining rules with no exceptions since, for every exception to a rule, the language becomes more complicated in terms of understandability and implementation. The minimization of defining rules without exceptions also contributes to the power of the language, since every exception makes the language less powerful in that it introduces a restriction. The expressive power, therefore, comes from ensuring that the composition rules are complete and minimal with no exceptions.

In persistent systems, there are three design rules which have their origins in the above work. They are parsimony of concepts, orthogonality of concepts, and completeness of the computational environment. The combination of these three rules yields the integrated persistent systems alluded to earlier.

Constructs are required that match well the application domain. The required parsimony of concepts focuses attention on to those essential constructs. The absence of extra features ensures simplicity. Orthogonality ensures that the system is composed of atoms, including persistent data, that may be combined in powerful ways to yield the appropriate abstractions. The case for completeness is that, if the language is not computationally complete, then the implementors and designers will need to combine the language with other facilities to build a PAS. This reintroduces the problems of disharmony.

Constructing persistent systems is made considerably easier when the whole computational environment is persistent. In such an environment, programs and processes may be regarded as data, and manipulated in the same manner, allowing transformations traditionally regarded as being performed by a separate mechanism to be executed within the persistent environment. For example, in the case of procedures, the persistent store may be used to provide a uniform library structure in a manner similar to providing a library of data parts (Atkinson and Morrison, 1985, Atkinson et al., 1993*a*, Kirby et al., 1994*a*).

2.2 Persistence Principles

The general principles presented above lead to specific principles concerning persistence.

2.2.1 Definition of Persistence. The persistence of a data object is the period of time for which the object exists and is usable (Atkinson, et al., 1983*a*). We aspire to systems where the use of data is orthogonal to its persistence.

2.2.2 Orthogonal Persistence. There are three principles of persistence that may be used to achieve the above design goals. They are:

> *The Principle of Persistence Independence.* The form of a program is independent of the longevity of the data which it manipulates. Programs look the same whether they manipulate short-term or long-term data.
>
> *The Principle of Data Type Orthogonality.* All data objects should be allowed the full range of persistence, irrespective of their type. There are no special cases where objects are not allowed to be long-lived or are not allowed to be transient.
>
> *The Principle of Persistence Identification.* The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. The mechanism for identifying persistent objects is not related to the type system.

The application of these three principles yields *Orthogonal Persistence.*

2.2.3 Persistence Independence. Persistence independence frees the programmer from the burden of having to explicitly program the movement of data among the hierarchy of storage devices, and from coding translations between long-term and short-term representations. As an example of persistence independence, consider a sorting procedure that takes as input an array of objects to be sorted. The parametric array may be small or large, and is held in main store or disk. Beyond having to identify the correct array, the programmer need not be concerned with the size or storage details of the arrays. Thus, the user does not have to, indeed cannot, program to control the movement of data between long-term and short-term store; this is performed automatically by the system. The mechanical cost of performing the movement of data does not disappear but the intellectual cost does. That is, the programmer need not specifically write code for it, making the application code smaller and more intellectually manageable. The implementor of the support system now has the challenge of automating that data movement and any translation efficiently.

2.2.4 Data Type Orthogonality. Data type orthogonality is an aid to data modeling in that it ensures that the data model can be complete and independent of the persistence of the data being modeled. For example, bulk data types abstract over size, and are therefore commonly used in persistent programming languages to aid the manipulation of massive collections of data such as scanned data from satellites or insurance policies sold by a company. Where such data are only considered long-term, the data model has to allow explicit conversion between long and short-term forms to allow creation of new bulk data and the manipulation of extracts from the long-term bulk data as short-term data.

Data type orthogonality also includes the language design principle of type completeness. For example, for bulk data this means that the elements of the bulk constructor are independent of the persistence of the data. Thus, the programmer is not faced with a system where a set of one particular element type is allowed to be persistent and another element type is not.

2.2.5 Identification of Persistent Objects. A number of methods have been investigated to identify persistence of data. Some involve associating persistence with the storage allocator, variable name, or the type in the declaration. Under the rule of persistence independence, these are disallowed. They are also inappropriate for other reasons (Atkinson et al., 1986).

In languages where the extent of a data object can differ from its scope, a limited form of persistence already exists. Such objects are normally kept on a heap, and whether they can still be used depends on the availability of legal names. That is, they are kept available as long as the program and execution environment contains enough information to refer to them. This limited persistence can be extended to allow objects to persist beyond the activation of the program.

The technique that extends the above and is now widely used to implement this principle is *identification by reachability*. In this, the identification of persistent objects is performed by the system automatically by computing the transitive closure of all objects reachable (by following pointers) from some persistent root or roots (Atkinson et al., 1982; Atkinson et al., 1983*b*; Cockshott et al., 1984; Brown and Cockshott, 1985; Brown and Morrison, 1992). Such data then can persist over activations of the programs that operate on them. The analogy with automatic garbage collection is obvious.

2.2.6 Loss of Orthogonality. Loss of orthogonality of persistence occurs by disregarding any of the three principles given in Section 2.2.2. Most serious is persistence independence, because, if it is broken, it is hard to see how the persistence in the system is orthogonal in any way.

Some programming language designers have taken a very pragmatic approach to persistence, and disregarded data type orthogonality for specific types. Pascal/R (Schmidt, 1977), DBPL (Schmidt and Matthes, 1992), and RAPP (Hughes and Connolly, 1990) are persistent programming languages based on Pascal where only first-order relations that contain no pointer types may be persistent. This corresponds to relational database practice, but causes difficulty when other modeling techniques are used. The restrictions imposed are not necessary, and it has been shown elsewhere that persistence can be added orthogonally to Pascal (Berman, 1991).

Figure 5. Persistent objects before being sent to the persistent store

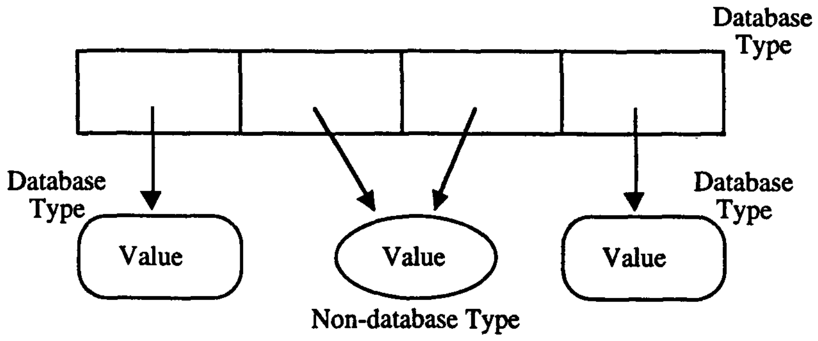
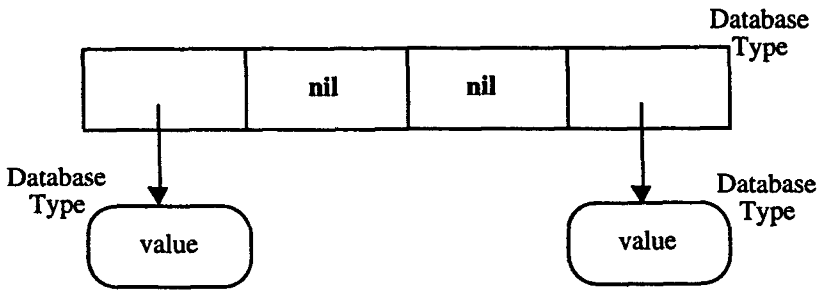


Figure 6. The same objects as in Figure 5 in the persistent store



Where reachability is not used, languages often associate persistence with type. This immediately violates the principle of persistence identification and, as a side effect, the other two principles as well. It also gives rise to dangling reference problems, or at least invalidated references for persistent objects that point to non-persistent objects. The E programming language (Richardson and Carey, 1989; Richardson and Carey, 1990), most persistent extensions to C++, and the PGraphite language (Wileden et al., 1988) all use this technique. Figures 5 and 6 illustrate this problem.

Figure 5 shows an object before being preserved in the persistent store. It has a type that permits its residence in the database, and it holds two references to other objects whose types are such that they also may be stored in the database, as well as another value that has a type for which database residence is not supported.

Figure 6 shows the result of saving the structure in Figure 5 in the persistent store. Only some of the structure is preserved, and the remaining, non-database value is lost, having been replaced by a dangling references or *nil* values if they are available.

Here, the persistent store model of data is not consistent with the main store model, a position we wish to avoid, since this adds complexity to the system for the programmer to master. For example, to overcome this loss of data when data are required to exist between program executions, programmers would need to do the following:

1. define a surrogate structure to the one shown that only uses types that are supported by the database;
2. write code to translate the non-database values to equivalent database-values in this surrogate; and
3. write code to perform the inverse translation.

They then would have to ensure that this code was run at appropriate times, and that the combined translation was exactly consistent.

The lack of completeness in the persistent computational environment can be seen in some C++ OODBMS. In these, the methods are held in a traditional program library and linked into C++ programs using a standard linker. The data values of the objects are held in the database. Garbage collection of unwanted data now becomes a problem in this model, since it is not known whether methods contain references to objects, or whether any objects still exist that require the methods.

Some systems do not provide garbage collection of the data at all: a solution which eventually will lead to grief. Other systems allow explicit deletion of an object or recursive deletion from a root. In either case, methods still may exist in the program library which now do not have any data; this usually causes run-time failures.

2.3 Savings with Persistence

The benefits of orthogonal persistence have been described extensively in the literature (Atkinson, 1978; Atkinson, et al., 1982, 1984; 1985, 1988*b*; Atkinson and Morrison, 1985; Morrison et al., 1985, 1986, 1987*b*, 1993*a*, 1995; Atkinson and Buneman, 1987; Dearle and Brown, 1988; Brown, 1989; Connor et al., 1990*a*, 1993; Cooper, 1990*a*, 1990*b*; Albano et al., 1993). They can be summarized as:

- improving programming productivity from simpler semantics;
- avoiding ad hoc arrangements for data translation and long-term data storage;
- providing protection mechanisms over the whole environment;
- supporting incremental evolution; and
- automatically preserving referential integrity over the entire computational environment for the whole life-time of a PAS.

The first saving of persistent systems is in the reduced complexity for application builders. Traditionally, the programmer has had to maintain three mappings among the database model, the programming language model, and the real world model of the application (Figure 3).

The intellectual effort in maintaining the mappings and overcoming the complexity of the support system distracts programmers from mastering the inherent complexity of the application. In a persistent system, the number of mappings is reduced from three to one (Figure 4), thereby simplifying considerably the tasks undertaken during PAS maintenance and construction.

Corresponding to the intellectual savings of persistence, there is also a saving in the amount of code required to maintain the mappings. This has been estimated to be at least 30% of the total code for a typical database application (King, 1978). The unnecessary code is concerned with the explicit movement of data between main and backing store, and that required to change the representation of the data for preservation and restoration. An example of the former is input and output code, and of the latter is code to flatten and reconstruct a graph before output and after input. Thus, the size of the application code is reduced thereby producing savings throughout the software life cycle of the application.

The third benefit of persistent systems is that a single model of protection may operate over the whole environment (Morrison et al., 1990*b*). In most programming languages, the simplest way to break the protection system is to output a value as one type, and input it again as another. Thus, security is lost over the persistent store. Using a single enforceable model reduces complexity, while increasing the protection as its type-checking prevents disallowed operations and hence misuse.

Database programmers often build large systems by incrementally designing subsystems and, then, by adding data and schema to the existing system. Initially, a schema is defined, then the database is populated with data and, finally, the programs (queries) are added. This is a powerful software engineering technique—that of incremental design and implementation. In persistent systems, the paradigm is preserved but with rather more flexibility in the roles of the schema, data, and programs. In particular, the schema is no longer considered to be relatively static and small, with the data and programs being dynamic and large. In persistent systems, the schema, data, and programs are all considered equally and may be static or dynamic, large or small, depending on the PAS under construction.

The final saving with persistence is that referential integrity of objects is automatically enforced. The referential integrity of an object means that, once a reference to an object in the persistent environment has been established, the object will remain accessible via that reference for as long as the reference exists. Furthermore, the identities of the objects are unique, and comparison of identity yields the same result independently of when it is performed. In a strongly typed persistent environment this also means that the type correctness of all such references is maintained (i.e., once a reference has been established, the type of the object referenced will not change). As will be seen later (Section 5.2), this has a number of consequences for using references in source descriptions in PASs.

The provision of orthogonal persistence has several consequences for support systems. For example, much that is done explicitly by programmers in a non-persistent system has to be automated by the implementors of persistent support systems. The abstraction over longevity also requires the accommodation of schemata, programs, and data of any size, since these may grow unpredictably during the lifetime of a PAS. An important aspect of such systems is the mechanisms that are used to construct sub-systems. Because of the reliability of control that is guaranteed by the persistent environment, such control can be delegated to tools without loss of safety (Connor et al., 1994*b*).

Considerable research has been devoted to the concept of persistence and its application in the integration of database systems and programming languages (Atkinson, 1978; Atkinson, et al., 1983*a*). A number of persistent systems have been developed including: PS-algol (Atkinson, et al., 1983*a*; PS-algol, 1988), Abstract Data Store (Powell, 1985), Flex (Currie, 1985; Stanley, 1986; Stanley and Drummond, 1988), Galileo (Albano et al., 1985); Amber (Cardelli, 1986), Trellis/Owl (Schaffert et al., 1985), TI Persistent Memory System (Thatte, 1986), Tycoon (Matthes and Schmidt, 1992), Napier88 (Morrison et al., 1994*a*) and Fibonacci (Albano et al., 1995). The underlying goal in this work is to achieve engineering that is of sufficiently high quality to ensure that the orthogonal persistence abstraction performs well, irrespective of the longevity and type of data and programs, so that application programmers can program without having to think about data transfers and similar issues.

3. Integration Concepts

The integration of programming languages and databases into a single computational entity involves a number of challenges for researchers in identifying and unifying the concepts used for similar purposes in each domain. Of particular interest are: the unification of type systems and data models, binding mechanisms, techniques for identifying persistent data, system architectures and concurrency control, and taking advantage of the controlled environment. Progress in these areas is discussed below.

3.1 Types and Data Models

The long-term goal of research into persistent type systems is to unify type systems and data models by developing an adequate model of type that meets the computational needs of persistent systems (Connor, 1990). Ideally, we would like a simple set of types, and a type algebra, so that by a succession of operations and the provision of parameters, any data model or conceptual data model can be defined. This we call the *type alchemist's dream* (Atkinson and Morrison, 1986).

Type systems provide both data modeling and protection facilities within databases and programming languages. Data modeling is performed in databases using data models, which have types to describe the form of the data, and in programming languages by using a classical type system. In both cases, the universe of discourse of the system is defined by the set of allowable types which, in turn, are denoted

Table 3. Equivalences between data models and type systems

Databases	Programming Languages
data models	type systems
schema	type expression
database	variable
database extent	value

by the set of legal expressions in the language. Data protection is provided by enforcing explicit and implicit integrity constraints in databases, and by type checking in programming languages.

As a first step in the unification of data models and type systems, some *approximate* equivalences can be recognized. These are summarized in Table 3. While the equivalences are only approximate, they do provide some insight into why integration may be possible at a conceptual level.

The issue of type checking is central to a type system that provides data modeling and protection for persistent systems. Generally, data models in databases are concerned with the manipulation of the data that is consistent with the constraints imposed by the data model. In some cases, these constraints may depend on values calculated during the computation. As such, they can be dynamic in nature, and require dynamic integrity constraint checking for enforcement. By contrast, classical type systems for programming languages are concerned with static checking, which allows assertions to be made and even proved about a computation before it is executed. Static checking therefore provides a level of safety within the system. It also allows more efficient code, since type checking code is not required at run-time.

At first, the dichotomy between the checking times in databases and programming languages appears to be beyond resolution. One way forward is to pursue the limits of static checking while still accommodating the dynamic checking required in particular instances. This retains the safety and efficiency of static typing provided in programming languages in most cases, while accommodating the dynamic flexibility of data modeling.

There are two approaches currently used to provide static checking for persistent type systems. The first is constraint specification where constraints over the data for a particular computation are expressed in some language. The checking requires a powerful theorem prover, sometimes beyond the limits of those currently available. Such systems are usually undecidable, and an unsuccessful check may be caused by the limitations of the theorem prover, rather than inconsistent constraints. However, where the theorem prover fails it may sometimes provide useful information that can be used to form minimal dynamic checks (Sheard and Stemple, 1989). This clearly delimits the points of dynamic checking while retaining as much static checking as possible.

The second approach is to extend classical type systems with specific types. As will be demonstrated later, points of dynamic checking are required to bind to persistent objects where programs and data are prepared separately and combined dynamically. The goal is to provide for some dynamic type checking while retaining as much static checking as possible.

Both of these approaches need to be pursued, but only the latter is presented here (Section 3.1.1).

One final difference is the semantics associated with the time of checking. In programming languages, the type checks, both static and dynamic, precede operations, whereas dynamic database integrity constraint checks may be performed after operations but before commitment. In all cases, however, the strength of the checking is not compromised by the time at which it is performed.

3.1.1 Persistent Type Systems. The major challenges for type systems for persistent programming are to provide the richness of expression of data models, and the completeness of protection required, in a mostly static form. These two goals are pursued in this section.

Programmer-Defined Type Constructors. An important innovation in type systems has been the provision of programmer defined type constructors (Milner, 1978; Cardelli and Wegner, 1985; Cardelli, 1989). This allows programmers to identify and introduce new and succinct notations corresponding to frequently used structures. Perhaps, more importantly, it also gives those structures a name that signifies their meaning to other programmers, thereby fulfilling one requirement of types or DDLs: to describe data effectively.

A sequence of examples of declarations is given to convince readers that programmer-defined types provide a convenient notation and an extensible descriptive system of comparable power to many data models.

The examples are written in a style close to the Napier88 notation (Morrison et al., 1989*b*; Morrison et al., 1990*a*) with the assumption that structural equivalence checking is in operation (Connor et al., 1990*a*). A new type name is introduced with a notation such as the following:

```
type Count is int
```

which declares `Count` as a type name, and makes it a synonym for the type **int** assumed already defined. Similarly, the following might define a `Date` type.

```
type Date is record (day, month, year : int)
```

These definitions become more interesting when type parameters are introduced:

```
type Pair [T] is record (first, last : T)
type DateStampedValue [T] is record (value : T; date : Date)
```

The first definition, `Pair[T]`, describes an infinite class of record types with two labels, first and last, both of the same type. The definition of Pair may be parameterized by a type to yield a concrete type such as Pair[int].

Strictly, these latest definitions are not types but type operators that, when parameterized by a type, yield a type constructor. We will make little of this distinction at the moment.

The second type definition illustrates how a type defined earlier may be used in another definition to build up a succession of more complex (more semantically meaningful) names of structures. In this case, DateStampedValue will generate record types that combine a date stamp and a value, for example:

```
type DateStampedImage is DateStampedValue [image]
type DateStampedCount is DateStampedValue [Count]
```

declares the type DateStampedImage, whose instances would hold images and the date on which they were captured, while instances of the type DateStampedCount would hold Counts and the date on which the count was made.

As a further illustration, the next two statements define co-ordinate types in integer and real spaces.

```
type IntXY is Pair [int]
type RealXY is Pair [real]
```

Regular structures such as those that occur in bulk types can also be defined. For example:

```
type Sequence [Element] is ...
type Ring [Element] is ...
type Set [Element] is ...
type Bag [Element] is ...
type Map [Domain, Range] is ...
type Tree [Key, Value] is ...
```

Readers have been spared the details of the definitions; typically, they would involve abstract data types as described in the next section. These definitions can themselves be composed.

```
type MultivaluedFunction [Dom, Range] is Map [Dom, Set [Range]]
type Forest [Key, Value] is Sequence [Tree [Key, Value]]
```

They then can be used with more conventional definitions.

```
type Student is record (matric: int; fname, sname: string;
    dateOfBirth: Date; ...)
type Course is record (title : string; prerequisites : Set [Course])
type Enrollment is record (student : Student; course : Course)
type Curriculum is record (student : Student; completed :
    Sequence [Course])
type Class is Set [Student]
```

All the type operator names defined above: `Pair, DateStampedValue, Set, Bag, ..., MultivaluedFunction`, and `Forest` can be used just like those supplied by existing data models or those supplied initially in the language. Similarly, the types and constructors that have been produced and named, vis: `Count,Date,DateStamped Image, DateStampedCount`, and `Class` can be used to produce instances just like types in a traditional data model.

It is, therefore, easy to envisage persistent support systems being shipped with few built-in types and constructors, but with extensive libraries of additional types and corresponding manipulation functions. Like traditional libraries of procedures, they then are available for those who want them. This gives a simple starting point for application builders, but a potentially rich repertoire of additions which can be augmented further when needs have been identified, either by persistent support system suppliers or by the application builders themselves.

Use of all of the parametric type constructors described in this section is made much easier if associated libraries of generic procedures are provided (Atkinson, et al., 1993*a*). This is possible only if the language provides appropriate forms of polymorphism such as those described below (Morrison et al., 1991).

Information Hiding using Existentially Quantified Types. The parameterized type operators introduced in the previous section define an infinite class of types. However, values can only be created with a specific type, and the information contained in them is available only to those parts of programs that have the correct type descriptions (i.e., those type descriptions that are structurally equivalent). Two values created using a type operator are type equivalent only if they are created using equivalent parametric types.

The type operator mechanism also can be used to provide information hiding or abstract data types. By encapsulating the structure of the data behind a published interface, the type of the data, and thus their internal structure, may be hidden. In consequence, *two abstract data types with equivalent interfaces are equivalent irrespective of their internal type (or structure)*. This is best illustrated by an example. We start from a record type defining counters as a value and an increment operation.

```
type Counter [t] is record (value : t ; inc : proc (t → t))
let intInc = proc (x : int → int) ; x + 1
let intCounter = Counter [int] (0, incInt)     !create a Counter
...
intCounter.value := intCounter.inc(intCounter.value)  !increment it
```

This creates a record with two fields, an integer `value`, and a function `inc` to increment the value. The type of the record is equivalent to

```
record (value : int ; inc : proc (int → int))
```

While the record is useful for aggregation, it does not protect the data from outside interference. For example, both of the fields may be accessed and perhaps updated by any other part of the system that can specify an equivalent type. The final statement in the above example shows the value being updated by using the fields consistently as intended. However, the type Counter does not restrict the use of the value to the function inc.

Second-order information hiding (Cardelli and Wegner, 1985) may be used to abstract over the type of the data and, thereby, the required consistent usage can be statically enforced. An abstract data type may be formed from a parametric type operator in the same manner as is used to create a concrete value. In this case, however, the parametric type (or witness type as it called in this context) once instantiated becomes abstract. Consider, for example, the following revised creation of a counter.

```
let absIntCounter = abstract Counter [int] (0, intInc)
```

This creates an abstract data type, absIntCounter, where the two fields both operate over the same type but, once the abstract data type is created, that type (the *witness* type) may never be re-discovered. As a consequence, only the functions within the abstract data type may operate on the value, for they are the only ones with the witness type. Since the witness type is abstracted over, but is known to exist, the type of absIntCounter can be written more formally as

```
        ∃t.record (value : t ; inc : proc (t → t))
```

A second advantage is that all abstract data types formed in the same manner are type equivalent. For example,

```
let absRealCounter = abstract Counter [real] (0, incReal)
```

also has the type

```
        ∃t.record (value : t ; inc : proc (t → t))
```

since the witness type is abstracted over. These types may be used as follows, where absCounter is the type of all Counters as written in the above two examples.

```
let updateValue = proc (aCounter : absCounter)
    aCounter.value := aCounter.inc (aCounter.value)
...
updateValue (absIntCounter)
updateValue (absRealCounter)
```

One final problem remains for abstract data types. To preserve static type checking, it is necessary to create a context to use the components of the abstract data type. The necessary restriction is that it must be possible to identify by static analysis that, when the interface components are used, they are only used in a manner that ensures that they all belong to the same abstract witness. Once this has been achieved with an appropriate syntactic mechanism, the requirement that the hidden information only be used consistently can be enforced by static analysis of the program text during type checking. This provides safety, since it allows programmers to place precise limits on the ways in which long-lived data may be used. The details of models of witness types can be found in Cardelli and MacQueen (1988).

Generic Re-usable Persistent Polymorphic Procedures. The introduction of a statically typed universe trades the power of expression for static safety. How much the system designer is willing to trade depends upon the application domain.

In Albano et al. (1989) and its companion articles, an analysis of what constitutes a persistent type system is given. For modeling purposes, it is generally agreed that some form of polymorphism is required to capture the expressiveness of data models, and to increase component re-use (Morrison, et al., 1987*b*) as has been demonstrated already by abstract data types. The most favored forms of polymorphism are universal polymorphism: parametric or inclusion.

Parametric polymorphism describes the polymorphism found in ML (Milner, 1978) and its derivatives, whereas inclusion polymorphism is the style of polymorphism found in object-oriented languages such as Simula67 (Dahl and Nygaard, 1966). An interesting hybrid may be found in the database programming language Galileo (Albano, et al., 1985), which is a derivative of ML but utilizes inclusion polymorphism to implement part of the Semantic Data Model (Hammer and McLeod, 1981). Cardelli and Wegner (1985) have shown separately how the parametric and inclusion forms of polymorphism may be integrated, as bounded quantification, to yield forms of abstraction not available to either one.

Since parametric and inclusion polymorphism give similar power (Cardelli and Wegner, 1985), only parametric polymorphism is described here. In parametric polymorphism, functions may be defined that work for all types. Consider a function that counts the number of elements in a list. Since it does not use the values in the elements of the list itself, then the function should work for all lists. It is said to be universally quantified and its type may be written as:

$$\forall \texttt{t.(List [t]} \longrightarrow \textbf{int}\texttt{)}$$

To extend this example, consider functions that maintain an index that would store values of any type, and use keys of any type that has (at least) equality defined over it. The functions therefore must be polymorphic over the value and key types. This is specified relatively easily using parametric polymorphism. For example, if the index is implemented by a list of pairs, one element for the key and one for the value, then the type of the function to enter a value in the index is

$$\forall \texttt{Key.}\forall \texttt{Value.(Key}\times\texttt{Value}\times\texttt{List [Key}\times\texttt{Value]} \longrightarrow \texttt{List [Key}\times\texttt{Value])}$$

The advantage of this type of abstraction is that the functions can be placed in the persistent store and used later to operate over indexes of all types. The software engineering advantages of writing concise code that is generally applicable are well known.

Further abstraction can be obtained by observing that the functions should be able to abstract over the structure of the data, as well as its base type. The above function can be written, since the polymorphic procedure heading specifies that the index is implemented by a list. Thus, it may be implemented using the operations on a list. It is possible to implement the index using other structures such as arrays or B-trees. For example the type using a B-tree is:

```
∀Key.∀Value.(Key×Value×Btree [Key×Value] → Btree [Key×Value])
```

Combining the functions requires that the parametric types be equivalent. Abstract types also may be used to yield a combination of universal and existential quantification. Note that, with universal quantification, one abstract polymorphic form can be written and special cases generated whereas, with existential quantification, existing values are described by a more general type, allowing more general abstraction over that type.

Polymorphism is one technique for retrieving some of the expressive power that is lost by the introduction of the type system in the first place. By abstracting over the types, more general computations can be expressed than in monomorphic systems. Polymorphism does not regain all of the power lost by the introduction of a statically checked type system. Relational systems are highly polymorphic, but depend upon the dynamic evaluation of types for their expression. For example, the specification and implementation of a generic natural join function provides an example of abstraction over types that is beyond the capabilities of most statically typed polymorphic type systems. This is because the details of the input types, particularly the names of the tuple components, significantly affect the algorithm and the output type.

Type systems that can accommodate functions such as natural join are generally higher order and dynamically checked. An interesting exception in this case is the Machiavelli type system (Ohori et al., 1989). In general, however, to retain static checking and still write these higher-order functions, a different technique is required. One such technique is linguistic reflection (Stemple et al., 1992*b*), discussed later in this paper.

Persistent Type Checking. We first demonstrate the need for some form of dynamic type checking in persistent systems, and then show how it is provided.

The novelty of a persistent type system is that it must provide type checking over the whole of the computational activity, including the use of persistent data. Within a traditional file system, the use of persistent data is achieved by programs having textual descriptions of how to find data within the file system. For type checking, a static description of the file data within the program is used by the compiler to assert that the data will be of the specified type when it is provided from the file. For implementation, the run-time system must perform dynamic type checks to enforce these assertions. Figure 7 illustrates the point.

In Figure 7, a type `Address` is declared to be a record with three fields `name`, `age`, and `gender` of types `string`, `integer`, and `boolean`, respectively. A variable

Figure 7. Dynamic file access

```
type Address is record (name : string; age : int; gender : bool)
var fileDescriptor : file [Address]
fileDescriptor := open ("/user/ron/addresses")
! At this point the system accesses the file ensuring that it
! exists and checks that the type of the data in the file is
! of the specified type
if fileDescriptor = nil then error ... else
begin
    while ~eof (fileDescriptor) do
    begin
       let this = read (fileDescriptor)
       ...
    end
end
```

fileDescriptor is declared as having the type of a file of Address. It is initialized to the file at path name "/user/ron/addresses" by the open statement. If the file exists, then a dynamic type check is performed to ensure that the data contained in the file is of the correct type, namely Address. Note that the interpretation of the path name string during the open operation is an example of the failure of computational completeness described above, as it is not determined by the semantics of the language.

It should be noticed that in Figure 7 there is only one point of dynamic typing (i.e., during the open statement). The program is compiled with the assertion that the file will have the correct type, and all other type checking, therefore, can be performed statically. Thus, the program may read values from the file without performing subsequent dynamic type checks. In this form of persistence, each file can be considered a persistent root. Notice also that the type of the values in the file must be guaranteed by the file system. Thus, overwriting the data with values of a different type is forbidden for a strongly-typed file system.

The nature of the open statement in Figure 7 is worthy of closer inspection. The statement has different types depending on the values contained within the file. In the above case, it returns the type file of Address but, in another case, may return, for example, the type file of Employee. The most general type for open is one that can be coerced into the specific type dynamically, during the open statement. Thus, open can be said to have a dynamic type.

Changing the type of the values in a file entails deleting the file and replacing it with a file of a different type. This appears to the program as if the old values have been coerced to a new type. Notice however that the coercions all are performed as a side effect of another operation, and that there need not be an explicit dynamic type or coercion facilities within the language. In a typical implementation, for example an Ada APSE, these operations depend on two systems with independent semantics, which can be changed independently. A *deus ex machina* therefore may change the type or existence of the file under the feet of the programmer.

As the number of typed files in the persistent space grows, programs will inevitably wish to access more than one file, and to have inter-file references. This is the major difference between a typed file store and a typed object store; the files are self contained with no inter-file references, whereas objects are highly inter-connected. Let us assume that typed files could be extended to accommodate inter-file references. As a consequence of inter-connection from a file, that file's type must have knowledge of the types of all the files to which it connects. Over time, the persistent space becomes highly connected necessitating every program to have a near complete description of the types of all the files.

The second problem with the typed file approach to persistence is that the file types are fixed at file creation time. Files cannot refer to newer files whose type has not yet been determined, which restricts evolution of the system. At a programming level, inheritance partially overcomes this by yielding additive evolution (Atkinson et al., 1993), which allows values to be replaced by more specialized forms. Thus, in Figure 7, the file could be one that provided an extra address field, such as `postcode`. The processing system must be set up so that it ignores this to achieve correct execution. To accommodate evolution completely, it is necessary to allow arbitrary changes in the data model, at least at some well defined points.

Both problems outlined above—partial specification of the type structure (schema) and the evolution of the data model—can be solved by the same mechanism that performs the dynamic type checking of persistent data. The integrated solution involves an infinite union type, of which type **pntr** of PS-algol (1988), **dynamic** of Amber (Cardelli, 1986), **any** and **env** of Napier88 (Morrison, et al., 1994*a*) are examples. Persistent roots have the infinite union type, and values are injected into these objects with a type, and projected dynamically onto a type for re-use. It is at the points of projection that the dynamic type checking occurs. Figure 8 illustrates this point using type **any** and the Napier88 notation.

In Figure 8, the standard function *PS* is called. It returns the value of the persistent store, which has the dynamic type **any**. The use of a dynamic type allows the persistent store to change its specific type between activations of particular programs. To use the value with its most specific type, *ps* is projected by the **project** clause onto that type. It is during the projection that the dynamic type check occurs. Within the projection context, the value may be used with its most specific type, in this case *Address*. The above program writes out the *age* field of the record. Any number of types may be specified in the projection; the first correct one is used. A catch all **default** clause is used to trap all the cases where none of the specified types match. Within this context the value only has the type **any**.

Figure 8. Projection from an infinite union

```
type Address is record (name : string ; age : int ; gender : bool)

let ps = PS () !This is the only standard function in Napier88
               ! It returns a persistent root of type any
               ! Before the value can be used with its specific type
               ! it must be projected onto that type


project ps as X onto
Address : write X (age)   ! X has type Address here
...
default :    ! This is a "catch all" and ps has type any here
```

Thus, the mechanism is that the persistent store has a most general dynamically checked type. To use the values within the store, the dynamic type must be projected onto the specific type for the store. The method by which the store may change its type, by injection of a value with a different type, is not of interest here, except for the fact that it can be performed.

This mechanism occurs implicitly in standard database interfaces. When a program opens a database, it specifies a schema or view. During the opening operation, the schema or view used during the program's compilation is compared with the database's current schema or view. Arbitrary changes may have been made to the database using a schema editor between the compilation and this execution. If the schema or view no longer matches the expectations established at compilation-time, then an error is signaled. Thus, internally the run-time system is able to treat the database as having a dynamic type, and to perform a dynamic verification that the expected and actual types match.

From Figure 8, it can be seen that dynamic checking is limited to the points of projection from the type **any**. Values of these dynamic types are first class and may also be a constituent part of any other type. Thus, a graph of objects in the object store may have one root of type **any** as well as other values of type **any** that are components of other objects. Before they can be used with their specific type they also have to be projected onto that type. This allows programs to specify only the part of the schema that they require up to a point of dynamic checking. As a consequence a schema, which is represented by an arbitrarily large collection of mutually referencing types, may scale well since the schema specification is bounded by the dynamic types. Incremental schema changes inject new values into an **any**, and the type of the rest of the specification remains unchanged. In addition, where an **any** encapsulates the type, type checking is postponed until required. Hence, excessive type checking costs on start-up are avoided (Figure 9). To implement this

Figure 9. Partial specification of types

```
type Address is record (name: string; age: int; gender: bool; extra: any)
let ps = PS ()
project ps as X onto
Address :
    begin
      let this = X (extra)        ! this is of type any
      type extraInfo is record (idNo : int ; spouse : Address)
        ! Programs not using the extra field do not need to specify
        ! extraInfo
      project this as Y onto
          ! type check using extraInfo only happens when this is
          ! executed
        extraInfo : write Y (idNo)    !write out the id number
        ...
      default : ...
    end
default : ...
```

delayed checking, it is necessary to store potentially large and complex types with the **any** value.

In Figure 9 where the *extra* field is used, the specification of the *extraInfo* type is required. Where programs do not use the *extra* field, the type *extraFieldInfo* does not have to be declared. Thus, only part of the type structure need be specified, the part of interest to the program (Figure 10).

The use of dynamic types allows the data model to evolve without recompiling all the programs that refer to the data. For example, if the *extraInfo* type is altered, then only programs that used that type need be altered.

To summarize, explicit dynamic types allow partial specification of the overall structure of the data (schema), and facilitate the evolution of the data, without having to alter programs that do not make use of the evolutionary changes.

Range of Type Checking Times. The addition of the infinite union type **any** facilitates incremental dynamic checking in persistent systems. Thus, the range of checking times includes just-before-use dynamic checking. At the other end of the spectrum, the presence of the persistent store allows persistent values to be bound to programs during program construction. Where the programs themselves are persistent objects, this leads to the concept of hyper-programming (Kirby et al., 1992*b*) that will be discussed later. For the present, it is sufficient to realize that persistent values are available to the type checker, allowing types to be value based. This leads to

Figure 10. Partial use of types

```
type Address is record (name : string ; age : int ; gender : bool ; extra : any)
let ps = PS ()
project ps as X onto
Address :   begin
                write X (name)     !write out the name
            end
default : ...
```

persistent systems being able to support some *dependent types* (i.e., types that require tests on values to establish their equivalence) similar to static constraints based on values in traditional database systems.

To introduce the usefulness of dependent types, a motivating example of dependent bulk types taken from Connor, et al. (1993) is given. The semantics of many bulk type data models depends on user-defined attributes such as definitions of element equality, ordering, and other domain predicates. While these attributes are an intrinsic part of the data model, they are not normally treated as part of the type description. This may lead to the occurrence of data modeling errors, such as a union operator accidentally being applied to two sets that have different semantics for the equality of their elements.

Directories indexed by Scottish names provide an example where the inclusion of element attributes in the type may be a requirement. People may take a different view as to whether "MacFarlane," "Macfarlane," and "McFarlane" are really different names, but their owners are usually protective of their different forms. If they are used to retrieve data, however, it is unlikely that the retriever will wish to distinguish among them. Therefore, there is a requirement for different types of directory, dependent on the domain equality semantics. For two directories to have the same type, they must have (depend on) the same equality operator, which is a value. If it is a persistent value, then they can depend on the same value, even if they were formed on different occasions.

For dependent types to be structurally equivalent, it is necessary for the values upon which they depend to be equal. For the equivalence of dependent types to be statically decidable, it is necessary for the values on which the types depend to be statically available to the type checker.

The essential requirement is that any values on which a type depends are evaluated before any equivalence testing is performed on the type. This restriction of the general type description can be enforced by restricting the dependencies to be existing persistent values.

A full description of such a language mechanism, including the explanation of the necessary restrictions and an introduction to the subject of polymorphism over dependent types, is given in Connor et al. (1993).

3.2 Range of Binding Mechanisms

Binding mechanisms present the user with a trade-off between safety and flexibility (Morrison et al., 1987*b*, 1990*b*). Dynamic binding is the most flexible, since the binding is delayed until the latest possible time at which a choice can be made. In contrast, static binding is safer, in that static checking may be used to give advice to a person better able to understand it (a programmer) and to eliminate run-time binding errors. The programmer has to choose the mechanism most suitable for a particular application from the range of binding available within the construction system. Persistent systems extend the range of binding by presenting the user with the possibility of using persistent objects during the construction of an application.

Traditionally, in programming languages and database systems, a binding occurs between a name and a value (Strachey, 1967). That is, a value is bound to a name for some period during the evaluation of a program or query. This has been extended by Burstall and Lampson (1984) to include a type, and further by Atkinson and Morrison (1987) to mutability. A binding mechanism, therefore, has four components: a name, a value, a type, and an indication as to whether the value is mutable or not. To complicate the issue further, bindings may be performed statically by the compiler or dynamically by the run-time system or, as is often the case, at intermediate stages.

3.2.1 Nature of Binding Mechanisms. As indicated above, the binding mechanism has four components:

- Is the binding to a mutable (L-value) or immutable (R-value) value?
- When is the binding performed?
- What scoping is involved?
- When is type checking performed?

Bindings may be made to immutable values (i.e., constant values that do not alter during the period of the binding), or to mutable values (locations) where the binding does not change, although the value referred to by that location may change. These kinds of bindings are traditionally known in programming language parlance as R-value (for immutables) and L-value (for mutables) bindings (Strachey, 1967). The manifest constants of BCPL (Richards and Whitby-Strevans, 1979) or Pascal (Wirth, 1971) are examples of R-value bindings. On the other hand, Pascal variables are examples of L-value bindings where the compiler may bind the name to a location, but not to a particular value since that may vary at run-time.

A binding is always performed with reference to a particular environment. That is, a binding will be part of a particular environment and may use other bindings in that environment to establish its own. The scope of the binding determines where it may be used. There are two common forms of scoping in programming systems: static and dynamic. In static scoping, the scope of the bound name can be detected by static analysis of the program. Algol 60, and all derived languages such as Pascal and Ada, utilize a static scoping rule that allows duplicate bindings to be detected

Table 4. Categorization of binding

	Static R-value	Static L-value	Dynamic R-value	Dynamic L-value
Static Typing Static Scoping	1	5	9	13
Static Typing Dynamic Scoping	2	6	10	14
Dynamic Typing Static Scoping	3	7	11	15
Dynamic Typing Dynamic Scoping	4	8	12	16

statically. In dynamic scoping, the binding in scope is the one that was last defined in the dynamic evaluation of the program. Dynamic scoping can be seen in Lisp (McCarthy, et al., 1962), in the binding of file names, and in the segment binding mechanism of Multics (Organick, 1972) and all derived operating systems.

Type checking can be performed statically by a compiler or by the run-time system. Dynamic type checking occurs when the run-time system executes code to ensure that the data are of the correct type. This typically occurs even in so-called statically checked languages when external data are brought into the computation (e.g., in read statements) and, as has been shown earlier, during projections out of infinite unions.

A categorization of binding is given in Table 4.

There are 16 different methods of binding, based on the four binding choices given in Table 4. The most static form is a static R-value binding with static type checking and static scoping. The most dynamic form is a dynamic L-value with dynamic type checking and scoping. It is interesting to note that, even within one particular language, there is often more than one binding category. For example, in Pascal, category 1 describes **const** values, category 13, variables, category 15, variant projections (cases in Pascal) and category 16, file names.

In determining the appropriate binding mechanisms for a particular system, the designer is faced with the problem of balancing safety against flexibility in a manner similar to type checking. The safety in the system is derived from being able to say (even prove) something about the program before it runs (i.e., statically) to improve confidence that it is correct.

An aspect of static checking that is often overlooked in programming systems is that the source code then acts as better program documentation. If a compiler can statically check a program, then so can a programmer. Thus, statically checked programs have better documentation properties and, consequently, better cost properties throughout the life cycle of the programs.

3.2.2 Range of Persistent Binding Times. It may be argued that the whole spectrum of binding mechanisms is required in persistent systems to facilitate the needs of prospective programmers. From observations the uses of bindings are:

- the creation of new objects (and binding into the persistent store);
- the reuse of existing objects (program and data) by new objects;
- new combinations of existing and/or new objects; and
- incremental construction of objects.

For example, the programmer may wish to bind statically to a combination of objects in the persistent store, in which case the objects are assembled into a new object and bound together. On the other hand, the programmer may wish to obtain the latest version of the object by delaying the binding until the object is about to be used.

The presence of a persistent environment in which programs are compiled and where programs may refer to persistent objects gives rise to three new perspectives in binding. They are:

- the extended scoping of bindings;
- the need to break bindings; and
- the separation of names and values in persistent bindings.

The presence of a persistent environment in which programs are composed, compiled, linked, and run extends the scoping of bindings to persistent values. Where the source programs themselves are values, static R- and L-value bindings may be made in the program source to persistent values. Thus, the source programs may contain already evaluated values that have been placed in the persistent store. These are manifest to the program and, as such, belong to category 1 which, up until now, has not been used much in programming systems. The advantage of this style of binding has already been shown in dependent types and will be shown later to lead to the concept of hyper-programming: a new technology only available in orthogonally persistent systems (Kirby, et al., 1992*b*).

Persistent systems also highlight the need to break bindings, and to recreate them later or in some other context. Consider the problem of releasing systems. To ship a system from one environment to another, it is often sensible to ensure that the system fits in with the new environment. In Unix, this often involves redefining shell variables or late binding the system calls to the new environment. For example, an application that uses the Unix *malloc* system call is both unlikely and unwise to take its local copy of the command along when it is shipped to a new environment.

The environment of a persistent value includes the graph of values that can be reached from it, and shipping values from one persistent system to another may involve copying the complete graph. This usually is not desirable for the semantics of the value, the utility of the value in its new environment, or from an efficiency point of view. To accommodate this rebinding, bindings first must be broken, which is in itself a dangerous activity. One proposal for achieving such rebinding in a safe manner is that of Octopuses (Farkas and Dearle, 1993). This allows the bindings within a value to be broken, producing a wiring diagram of the broken bindings. This wiring diagram then may be rebound in the new environment when the value is moved. Since the wiring diagram is effectively an abstract data type, it may be programmed to achieve a number of varied rebindings. The mechanism is a special case of linguistic reflection addressed specifically to this problem.

Flexible binding is possible only where there is a separation of the concepts of names and values. For flexibility, a value may participate in many bindings, and have many names (aliases). More importantly, in persistent systems, the name space is separate from the value space, allowing the graph of persistent values that constitutes the persistent store to have any number of name spaces layered on it. Application-specific naming schemes are supported by this arrangement.

It should be noted that polymorphism allows the separation of values and types, that structural type equivalence allows the separation of names and types, and that dynamic typing allows the separation of names and values. This is another example of the power of orthogonal design.

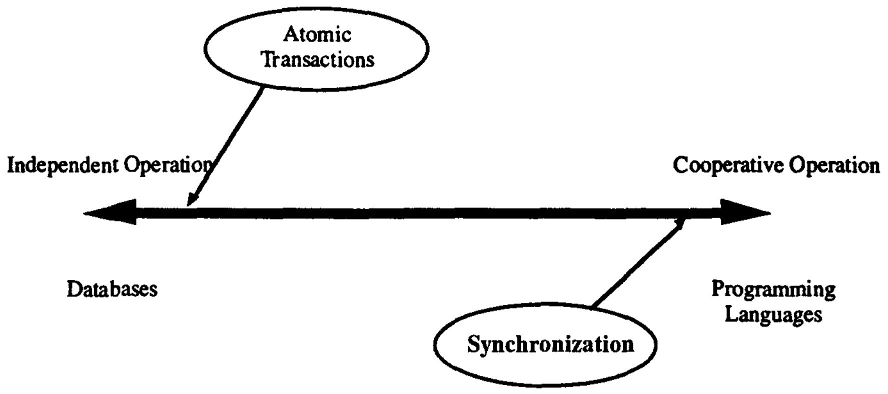
Some of the skill in using a persistent system will be in deciding which objects are statically composed and never changed, and which objects are dynamically composed. A judicious mixture of mechanisms has to be provided by the system, and eager binding is expected to be the preferred style of usage in that, when it is appropriate to perform the binding, then it is done for safety and not delayed unnecessarily.

3.3 Concurrency

Traditionally, the database and programming language communities have taken different approaches to concurrency control. In programming languages, concurrency control is based on the concept of the co-ordination of a set of co-operating processes by synchronization. Language constructs such as semaphores (Dijkstra, 1968*b*), monitors (Hoare, 1974), mutual exclusion (Dijkstra, 1968*a*), path expressions (Campbell and Haberman, 1974), and message passing (Brookes et al., 1980) have been provided to support this concept. By contrast, in databases, concurrency is viewed as a system efficiency activity that allows parallel execution and parallel access to the data. However, each database process may have to suffer the indignity of abortion to sustain the illusion of non-interference. The key concept in databases is that of serializability (Eswaran et al., 1976), which has led to the notion of atomic transactions (Eswaran, et al., 1976; Kung and Robinson, 1982), supported by locking (Eswaran, et al., 1976) or optimistic concurrency control methods (Kung and Robinson, 1982).

In both cases, the user must attempt to understand the computations in terms of some global cohesion. In programming languages, the emphasis is on synchro-

Figure 11. Spectrum of understandability



nization and the overall cohesion is understood in terms of the conflation of all the synchronizations. A number of techniques, including CCS (Milner, 1980) and the π-calculus (Milner, 1991), have been developed to help with this. In database systems, global cohesion is understood in terms of the concept of serializability (Eswaran, et al., 1976), but includes failure semantics such as abortion.

Figure 11, taken from Munro et al. (1994), illustrates a spectrum of understandability from the points of view of programming language and database users. It also illustrates that databases tend to use atomic transactions to enforce isolation, rather than co-ordinated sharing. Programming languages promote co-operation. Thus, in integrating databases and programming languages, the designer must unify these established and useful positions. However, the impetus does not come altogether from persistence, since languages that support atomic transactions, and databases that require non-serializable and designer transactions (Ellis and Gibbs, 1989; Sutton, 1990; Nodine and Zdonik, 1992) have been identified as necessary by their respective communities.

Both camps agree that co-ordinating a set of computations that share data is a complex undertaking, that it is difficult to characterize the power and behavior of the mechanisms, and that it is even more difficult to compare them with each other.

Previous attempts to provide concurrency in orthogonally persistent systems either have focused on a single model (Morrison et al., 1988; Morrison et al., 1989*a*), have provided a mechanism for extending synchronization, or have provided basic persistent threads. The work of CPS-algol (Krablin, 1987*a*, 1987*b*) is perhaps the most notable. CPS-algol added constructs to PS-algol to support processes and, hence, persistent threads. The concurrency model is co-operative based on synchronization by conditional critical regions. Concurrency is provided by executing procedures as separate threads. Using these primitives and the higher-order functions of PS-algol, a range of concurrency abstractions can be constructed including atomic and nested transactions as well as more cooperative models. This work has been revisited in Munro (1993) and Matthes and Schmidt (1994).

Given that the overall goal is for the user to understand the computations in terms of some global cohesion, the CACS system (Stemple and Morrison, 1992; Morrison et al., 1993*b*) proposes a mechanism for integrating concurrency control in programming languages and databases. It takes the point of view that the difficulty stems from both the low-level nature of the mechanisms and the inherent complexity of the problem.

The goal of the CACS approach is to control the coherence of sequences of operations on shared data in an understandable and flexible manner. The essence of the system is: understandability; the separation of concurrency control from data; formal capture; and a path to implementation. The global cohesion of an action in CACS is visualized as the movement of data among access sets. The visibility of the data to other users then becomes an issue of synchronization. Where the visibility coincides with commit time, atomic transactions may be obtained, and where the visibility is immediate, then synchronization is present. By viewing the coordination of the use of data as the coordination of the movement of the data among access sets, then the same data may be used in a different manner at different times. Thus, the data may take part in an atomic transaction one day, in a saga the next, and in a co-operative computation the next. A future goal of the work is to allow interaction of the concurrency control methods.

4. Technology to Support Persistence

An overview is now presented of some crucial aspects of the technology needed to support orthogonally persistence systems. The explanation below is not intended to be sufficient for someone trying to implement persistence, though the citations are intended to give an entry to the pertinent literature. It is intended to make readers aware of what is involved, and to convince them that orthogonal persistence is feasible.

4.1 Implementation Architectures

To support persistent applications, a stable and reliable store to hold the data and an execution mechanism to execute programs are required. The principal categories of supporting architecture are discriminated by the way in which these major components are provided and the way that they interwork. Four architectures are recognized in order of increasing commitment to the persistence philosophy. Not every reported system fits precisely into these categories, and their state of development varies between commercial products and research prototypes.

4.1.1 Combined Existing Systems. The designers may start from an existing data model (e.g., relational, object-oriented) and combine it with an existing language (e.g., C, Fortran, C++, Cobol). The combination proceeds by arranging a notation to perform bindings. This involves identification of types and data via databases, schemata, views and queries, and component use in programs or methods. The database component operations are made manifest by extending the language or providing some library.

The resulting combination may be sympathetic to the original language to varying degrees. Difficulties are often met because of the limited overlap between the two universes of discourse. Besides the type and conceptual differences to be overcome, syntactic noise is often introduced by attempting such a combination. There are, however, obvious advantages: capitalizing on training, utilizing well polished existing system code, continuing the use of legacy code or data with relatively little re-coding, and permitting multi-lingual working against the database.

Where the language and data model are reasonably sympathetic, an acceptable quality of integration can be obtained; for example, the integration of O_2 with C++ (Deux, 1990; Deux, 1991). However, it would be unrealistic with this architecture to expect that all the features of the programming language would work directly on persistent data and that data of all types could be stored. The storage of large collections of objects within a standard relational system is an interesting example of this approach (Reinwald et al., 1994). There, the programmer can use the full facilities of C++ to build and operate on an arbitrary group of objects and use the full relational facilities to store and retrieve such groups. However, programmers are not permitted to store persistent references between groups.

Implementations taking this approach attempt to leave both the language run-time system and the DBMS unchanged. Typically they are run as separate processes that communicate via messages.

4.1.2 Extended Existing Systems. This approach takes place in one of two directions. Designers may start with an existing database and extend it to have more complete type and computational facilities, or they may start with a language and add persistence. Postgres is a good example of the former (Stonebraker and Kemnitz, 1991). Commercial systems, such as Microsoft ACCESS (Microsoft Corporation, 1994*a*, 1994*b*) also fall in this category, as they start with a relational model, and expand its computational facility by adding program and interface generation facilities. The development of object-oriented facilities in the SQL3 standard (Kulkarni, 1994) corresponds to this approach.

The first version of PS-algol (Atkinson et al., 1983) was an example of this second approach, as it began by extending S-algol (Morrison, 1979). Similarly, Pascal-R was initially described as the addition of relational capabilities and long-term storage to Pascal (Schmidt, 1977). There are, however, fundamental differences between these two examples: the former set out to leave the language's type system unchanged and, hence, to achieve orthogonal persistence, the latter set out to extend the type system, but made no attempt at data type complete persistence.

Many persistent systems continue to be developed using this second approach. Typically, they are persistent versions of C or of C++ (Richardson and Carey, 1989; Shapiro et al., 1989; Richardson and Carey, 1990; Reinwald et al., 1994) but persistent versions of SmallTalk (Straw et al., 1984), ML (Matthews, 1985, 1989; Nettles and Wing, 1992) and Ada (Wileden, et al., 1988) have also been reported. Where the persistent language takes this form, there is the advantage of converting existing software to have persistent behavior with minimal effort.

Construction of a persistent system by extension of an existing system usually requires significant modifications to the original system's run-time system. Discussion here focuses on the case where a programming language is extended. There are four tasks to automate:

- obtaining the type information from programs, and accumulating it as a schema;
- storing, using a representation that will always be interpretable, any values that future programs may use, in a stable transactional store;
- identifying which values should be promoted to longevity, and detecting when they can no longer be used so that their resources can be re-used;
- arranging to load into a program's active store any persistent data that are about to be used.

Often the implementation involves new persistent store technology, including recovery, transactions, and space management. There are variations regarding the extent to which the store technology is tuned to the requirements of the particular language supported (Brown et al., 1992) and the extent to which the store builds-in clustering, indexes, bulk types, etc. (Cluet and Delobel, 1991; Matthes and Schmidt, 1991; Zezula and Rabitti, 1992; Cluet and Moerkotte, 1993).

Some persistent extensions to languages accept that they cannot implement all four of the tasks listed above. For example, the Texas store (Singhal et al., 1992) provides a stable checkpointable memory in which C and C++ programs run. In consequence, it provides orthogonal persistence for those languages, because the programs run unchanged in this store, but the representation would not carry data across changes to parts of the support system, and no automatic management of object lifetimes (task 3 above) is supported. These limitations are an ineluctable consequence of supporting any language that happens to be run in the store, as it means that there isn't a complete and reliable type system from which to derive the required information to provide these services.

4.1.3 Systems Based on an Integrated Design. Recently, several systems have been developed that are based on an integrated design, which tries to equitably use experience from both programming languages and databases, but starts afresh to develop a single system that provides the functionality of both systems. Examples are: Napier88 (Morrison, et al., 1994*a*), Fibonacci (Albano, et al., 1995), and Tycoon (Matthes et al., 1994) (all products of the ESPRIT Basic Research Action 6309: FIDE_2).

These generally have the advantage of being conceptually simple compared with the preceding approaches. However, they usually do not manage to provide full database facilities—that is, few can actually demonstrate a complete repertoire of incrementality, transactions, recovery, concurrency, distribution, and scalability. (It appears that this is more a consequence of teams being unable to muster the effort to tackle all of these issues together rather than of fundamental limits.) These support systems have to provide an interface to legacy code, as they are typically monolingual and have a unique data model or type system. Their implementation still has to achieve the four tasks identified above.

4.1.4 Persistent Worlds. All three of the categories discussed above are typically implemented above standard operating systems (e.g., UNIX, Windows NT). This has the advantage that it is easier to export the research to other sites. It has the disadvantage that it is extremely difficult to obtain entirely consistent behavior and reasonable efficiency on these platforms.

This leads to an alternative line of research where a totally new computational platform is constructed. This may be a persistent operating system (on top of which, all languages achieve persistence automatically, and with which all data have consistent persistent behavior) or even new hardware architectures with associated operating systems. Typical of the former are: Grasshopper (Dearle et al., 1994), EOS (Gruber, 1992, Daynès and Gruber (1994) (also a FIDE_2 product) and of the latter are: Rosenberg's (1990) proposals, and DAIS (Russell et al., 1994).

The pay-off from investing in good persistent technology is apparent only when a large application is used over a long period. Hence, it is difficult to properly evaluate this technology without further investment in building up a realistic load, writing a reasonable volume of application software, and conducting the evaluation over a long period (Atkinson, 1992, Atkinson et al., 1993*b*). Currently, none of the systems that build up persistence from the operating system kernel have received enough developmental effort to enable them to support such experimentation.

4.2 Implementing Persistence by Reachability

Support systems conforming to the last two architectures, and many that are essentially extended systems, implement:

- persistence by reachability;
- automated data movement; and
- re-use of space.

The basic algorithms for these three crucial tasks, which are applicable in all three architectures, are now presented. This is followed by a brief introduction to some of the ways in which the algorithms may be refined.

Table 5. Action taken on de-referencing a PID

1	discover whether the referend is already resident
2	if it is (false object-fault), overwrite the PID with the corresponding LA (swizzle [Moss, 1990; Wilson, 1990]), so that future de-references occur at normal computational speed
3	if it isn't (object-fault), read in the object, and keep a record of this PID to LA mapping to support steps 1 and 2, and then swizzle (i.e., overwrite the PID with the LA).

4.2.1 Incremental Loading and Swizzling. The persistent system provides some means of naming and loading persistent roots—in the kernel one root will suffice, since other naming schemes and multiple roots can be implemented with persistent data structures, and code accessed by this primitive root. The persistent root will contain references to other objects which will themselves contain references to further objects. Data movement into the active store (that against which code operates) is initiated when such references are de-referenced in an attempt to use the object to which they refer. These objects on which transfers are based may not correspond exactly to those which the programmers think about, as compilers may introduce mappings.

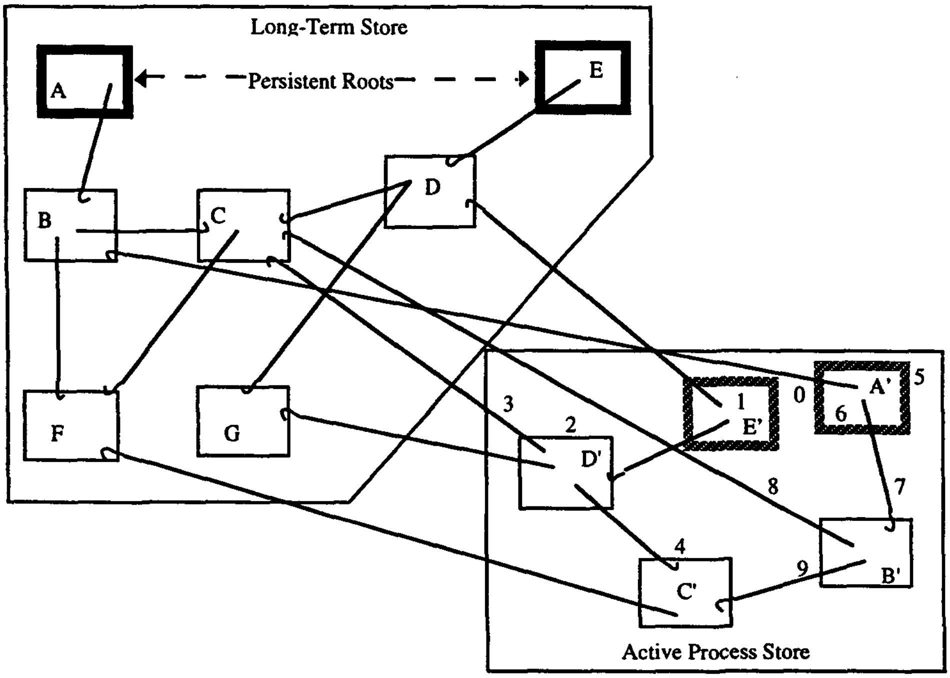
When a value is accessed, the support system arranges that it is loaded before computation is allowed to continue. Typically, there are two forms of address that refer to a value:

- a Persistent IDentifier (PID) that has long-term reliability, that is, it is some logical identification of the object, interpreted through a mapping or index, or (with less flexibility and permanence) it is a machine, device, and disk address; and
- a Local Address (LA), which is the form used by the executing process on the current hardware (e.g., a virtual memory address).

Different stores utilize different means of discriminating these two forms, and some operate with only one performing both roles; these issues are discussed later. It is desirable to arrange that, if a LA is encountered, the evaluation is not interrupted. However, if a PID is de-referenced (an attempt is made to use the value to which it refers), then the action in Table 5 is taken.

This algorithm will arrange that any long-lived data required by an application are automatically made available to that program. Note that, in step 3 of the above algorithm, it is necessary to record the relationship between a PID and its current local address in this process. This information is stored in a two-way accessible table, called here the PIDLAM (PID to Local Address Map). It is called the "Resident Object Table" in Kemper and Kossmann (1995). It has to be accessed by PID in

Figure 12. Lazy incremental loading algorithm



step 1 of the Table 5 algorithm to preserve sharing of substructures and by LA during de-swizzling (see Subsection 4.2.2).

Figure 12 is used to illustrate this automatic incremental loading process. It shows an intermediate store state during the execution of a persistent program. The boxes represent objects that may be in the long-term or short-term stores or in both. The lines denote references with the hook at the end corresponding to the referend. Thus, if we consider them attached to the object that holds them, the persistent objects are all those obtained by starting at each persistent object and pulling on all the hooks that emanate from those objects.

The relevant steps in the program execution that lead to the store state shown in Figure 12 are given in Table 6.

4.2.2 Promotion to Persistence on Checkpoint. Data flow from the active store to the long-term store also has to be automated. Application programs from time to time will request checkpoint, often as part of commit. At this point, any objects that have been brought from long-term store and then changed must be written back. In addition, new objects that are now reachable from the persistent root(s)

Table 6. Steps in incrementally loading data in Figure 12

0	Obtain persistent root E as E$'$
1	Attempt to de-reference PID to D—object-fault
2	Copy D as D$'$ into active store and swizzle: replace PID to D with LA of D$'$
3	Attempt to de-reference PID in D$'$ to C—object-fault
4	Copy C as C$'$ into active store and swizzle: replace PID to C with LA of C$'$
5	Obtain persistent root A as A$'$
6	Attempt to de-reference PID in A$'$ to B—object-fault
7	Copy B as B$'$ into the active store and swizzle: PID to B with LA of B$'$
8	Attempt to de-reference PID in B$'$ to C—false object-fault
9	Find PID of C from 4 and swizzle: PID to C with LA of C$'$

Table 7. Algorithm to promote objects to longevity

1	*Mutated* := {all the mutated persistent objects}
2	*Promotions* := {every new object directly reachable from *Mutated*}
3	*Promotions* := *Promotions* **union** {every new object directly reachable from *Promotions*}
4	**repeat** step 3 **until** no more additions
5	allocate PIDs for all objects in *Promotions*
6	transfer all members of *Mutated* **union** *Promotions* to long-term storage

must now be written to the long-term store, after they have been allocated space and a PID. As each object is written out, all of the LAs in the object must be de-swizzled, so that they now refer to PIDs. PIDs have to be allocated in advance of the transfers to correctly implement common sub-structures and cycles.

It should be noted that an object can be promoted to longevity only if some already persistent object has been updated to hold a reference to it, or if it is referenced by an object that itself is being promoted to longevity. The promotion algorithm is shown as Table 7.

The effect of this algorithm is illustrated with the aid of Figure 13, which shows part of the active store after a computation has continued from the incremental loading (Figure 12). It has mutated D$'$ and C$'$, and has created three new (transient) objects J, K, and L. It has overwritten the reference in D$'$ to G with a reference to J, which contains a reference to L, which contains a reference to B$'$. The object K references L but is not referenced by any persistent or newly formed object.

Figure 13. Part of the active store after mutation

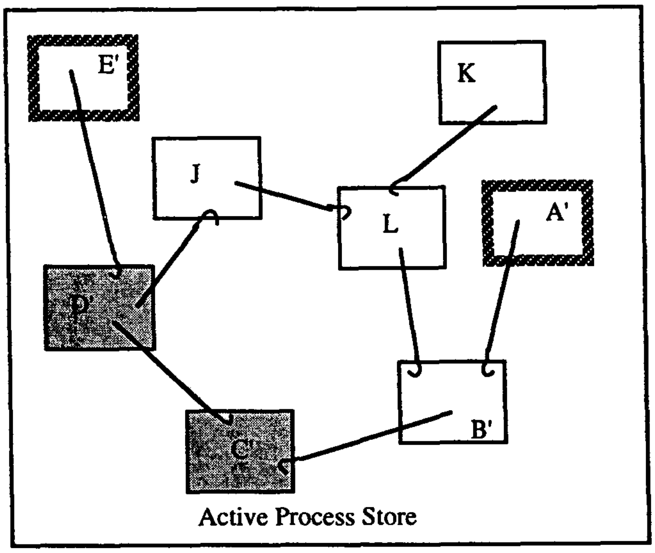


Table 8. Checkpoint of store in Figure 13

1	Discover *Mutated* = { D′, C′ }
2	Set *Promotions* = { J }
3	Set *Promotions* = { J, L }
4	Allocate PIDs for J and L
5	De-swizzle pointers in D′, C′, J and L
6	Copy D′, C′, J and L to long-term storage
7	If processing is to continue, swizzle pointers in D′, C′, J and L

The steps in the checkpoint on the store in this state are shown in Table 8. Carrying out these steps both efficiently and precisely without unduly slowing the execution before the checkpoint requires careful engineering. The copying in step 6 is usually performed in such a way that recovery is possible after a machine failure during checkpoint. For example, logs are written (Agrawal and De Witt, 1985; Moss and Sinofsky, 1988; Ruffin, 1992; Scheuerl et al., 1995) or shadow pages are used (Brown, 1987; Munro et al., 1994).

4.2.3 Garbage Collection and Object Termination. Persistence based on reachability implies that, when an object becomes unreachable, it is no longer useful—no future program execution can access it. However, it will still occupy space in the long-term store, and an opportunity exists to reclaim that space. An obvious method, also based on reachability, is garbage collection.

Since the long-term store is usually held on devices such as disk, and may grow very large, these garbage collections may be infrequent and incremental. The implementor trades space and perhaps density of useful data against garbage collection costs.

Garbage collection is traditionally used solely to recover space (Wilson, 1992). Garbage collections of the active store interact with the preceding algorithms in several important ways.

- When they fail to recover sufficient space by normal methods, they can release space by discarding unmodified objects that were previously loaded. This requires that pointers that once referenced these discarded objects be de-swizzled (a major issue in Kemper and Kossmann, 1995).
- When, even after discards of unmodified objects, there is still insufficient space, they can copy out modified objects or prematurely (and possibly falsely) promote new objects (steps 4 to 7 of the checkpoint algorithm must be applied). This increases the complexity of the subsequent checkpoint algorithms and is best avoided if at all possible.
- They can incrementally copy data to log files for recovery purposes (Kolodner, 1987; Kolodner et al., 1989).

These additional operations also may be invoked during garbage collection because the density of objects still in use has fallen to a level where thrashing has set in (Fenichel and Yochelson, 1969). Alternatively, dynamic clustering may be used to reduce thrashing (Benzaken et al., 1991).

4.2.4 Engineering Issues. Implementing the preceding algorithms with the support of conventional operating systems and hardware presents several challenges for store designers. A few of the choices are illustrated here.

The long-term form of the data may be exactly the same as that required for the active store. This avoids the costs and complexity of translation, swizzling, and maintaining a PIDLAM. It has the disadvantage of not scaling very well, and of not guaranteeing that the data (which normally include code) can be moved to a new platform as hardware architectures change (Atkinson, et al., 1993*b*).

If the same representation prevails then, for small, single-user systems, a total copy into virtual memory will be fast and adequate. For larger and shared systems of this kind, memory mapping techniques can be used (Koch et al., 1990; Rosenberg et al., 1990; Brown, et al., 1992; Singhal, et al., 1992; Munro, et al., 1994).

Object-faults (and false object-faults) can be detected by using the address translation unit's (ATU) protection or by in-line code before each de-reference. The former normally has the overhead of several transitions across the operating system boundary and the latter has costs even after the objects have been loaded.

Data movement can be on parts of objects, on objects, or on physical divisions (e.g., pages or groups of disk blocks) that match the hardware. The first has the advantage of accommodating very large objects, but has the disadvantage of increasing the complexity of object-fault detection (e.g., the next part of an image may need loading as a raster operation traverses it). The first and second have the advantage of not cluttering the working set with co-resident objects, but they require intermediate buffering and hence additional copying. Kemper and Kossmann (1995) showed that this has performance benefits. These two methods avoid phantom locking and checkpointing of co-resident objects, but complicate the implementation of resilience. The third method has the advantage that the actual transfer exploits the disk and channel well, and may even gain from optimizations made to support paging. It suffers because it brings into active memory co-resident objects that are not used. This detrimental effect will increase as a PAS ages unless a good dynamic clustering algorithm is operational.

Swizzling can be carried out at various times. Swizzling every time a PID is dereferenced has the advantage of avoiding the need to overwrite and later de-swizzle, but costs a look-up in the PIDLAM at every de-reference. This might be feasible if there were associative hardware supporting the PIDLAM (Russell, 1994).

Swizzling only on the first de-reference of a PID has the advantage of speed where algorithms repeatedly traverse the objects once they are loaded, but it involves updates to objects which may interfere with the use of hardware protection to discover the *Mutated* set or to record references to more recent generations (in generational garbage collectors (Cook et al., 1993).

As an object is loaded all its PIDs can be swizzled. Wilson uses memory mapping to do this a whole page at a time (Wilson, 1990). Wilson's scheme has the advantage of avoiding the costs of detecting any false object-faults, and it avoids system initiated writes to objects. It has the cost of allocating virtual address space more rapidly, and of restricting the variation between PID and LA to the page number part of the address, thus preventing objects from expanding as they are translated.

Translation of objects permits much longer longevity—objects can be held in a canonical form that does not derive from a particular architecture. Translation may be combined with decompression (on load) and compression on transfer to longer-term storage. It can be performed on a page or on an object at a time.

Discovering the *Mutated* set at checkpoint may be achieved by generating code during compilation that will insert objects into the *Mutated* set, or by using the ATU's protection system to detect an update dynamically. The former has the disadvantage that it incurs execution penalties on subsequent updates of the object (unless the compiler did some clever optimization) as in Moss and Hosking (1994). The latter has the disadvantage that it is only approximate (there are likely to be other objects in the same protected region) and interacts with other uses of the ATU's protection and the system initiated updates.

The above list is not exhaustive. Other major issues include whether to allow objects to span pages, whether to cluster and prefetch, where to allocate space and PIDs on promotion, whether to use shadow paging or logging, and whether a PID should encode a store location. Designing a good persistent object store is still a considerable challenge, as the interaction of this plethora of choices is only understood where it has been sampled.

4.3 Type-Safe Linguistic Reflection

Type-safe linguistic reflection is defined in Stemple et al. (1992*b*) as the ability of a running program to generate new program fragments, and to integrate these into its own execution. This is the basis for system evolution which itself is necessary to achieve adequate PAS longevity. For safety reasons, only strongly typed reflection will be considered.

Linguistic reflection has the goal of allowing a program's behavior to adjust dynamically to provide flexibility and high productivity. Thus, it extends the data modeling of the type system, and it should not be surprising to find a tension between type systems and reflection. The possibility that a program may significantly change its behavior decreases the opportunity for static type checking and, thus, compromises some of the benefits of typing. Thus, the reflective facilities are controlled in a manner designed to retain as much static type checking as possible without the control being so severe as to remove all the benefits.

The two techniques for type-safe linguistic reflection that have evolved are: compile-time linguistic reflection and run-time linguistic reflection. Compile-time linguistic reflection (Stemple et al., 1990; Stemple et al., 1992*a*) allows the user to define generators which produce representations of program fragments. The generators are executed as part of the compilation process. Their results are then viewed as program fragments, type checked, and made part of the program being compiled.

Run-time linguistic reflection (Dearle and Brown, 1988; Kirby, 1992*a*; Kirby et al., 1994*b*) is concerned with the construction and binding of new components with existing components in an environment. The technique involves the use of a compiler that can be called dynamically to compile newly generated program fragments, and a linking mechanism to bind these new program fragments into the running program. Type checking occurs in both compilation and binding.

The benefits of type-safe linguistic reflection in database and persistent programming consist mainly of two capabilities. The first is the ability to implement highly abstract specifications, such as those used in query languages and data models, within a strongly typed programming language. The second is the ability to accommodate some of the continual changes in data-intensive applications without resorting to ad hoc restructuring methods. Both capabilities involve reflective access to the types of a system that is changing itself and both approaches avoid introducing extra levels of interpretation.

Both compile-time and run-time reflection have been provided in previous languages. Compile-time reflection appears in the macro facilities of Scheme (Rees and Clinger, 1986) and POP-2 (Burstall et al., 1971). Run-time reflection appears in the *eval* functions of Lisp (McCarthy et al., 1962) and SNOBOL4 (Griswold et al., 1971) and the popval function of POP-2 (Burstall et al., 1971).

Type-safe linguistic reflection is different for the following reasons.

- More information is available to the reflective computation, in the form of systematically acquired types. This information can be used to automatically adjust to implementation details and system evolution. Without strong typing, linguistic reflection has little systematic information available about the structures involved in computation.
- The type safety of all newly generated program fragments is checked before they are allowed to be executed. Such type discipline is highly advantageous in a database programming environment in which the integrity of long-lived data must be maintained.

It is somewhat ironic that strong typing, which makes it difficult to integrate reflection with typed programming languages, is what makes linguistic reflection effective as an amplifier of productivity.

Type-safe linguistic reflection has been used to attain high levels of genericity (Stemple et al., 1990; Sheard, 1991) and accommodate changes in systems (Dearle and Brown, 1988; Dearle et al., 1990*a*); two examples of these are given below. It also has been used to implement data models (Cooper, 1990*a*; Cooper, 1990*b*, Cooper and Qin, 1992), optimize implementations (Cooper et al., 1987; Fegaras and Stemple, 1991; Cutts et al., 1994) and validate specifications (Fegaras et al., 1992; Stemple et al., 1992*a*). The importance of the technique is that it provides a uniform mechanism for software production and evolution. A formal description of linguistic reflection is given in Stemple et al. (1992*b*). Here, we concentrate on the uses.

4.3.1 Uses of Type-Safe Linguistic Reflection. Two examples of type-safe linguistic reflection are presented to show how the reflection mechanisms appear at the level of a programming language. The examples are *abstraction over types* and accommodating *evolution in strongly typed persistent systems*.

Abstraction Over Types. As indicated earlier, a generic natural join function provides an example of abstraction over types that is beyond the capabilities of most polymorphic type systems. The details of the input types, particularly the names of the tuple components, significantly affect the algorithm and the output type of the function, determining:

- the result type;
- the code to test whether tuples from both input relations match on the overlapping fields; and
- the code to build a relation having tuples with the aggregation of fields from both input relations but with only one copy of the overlapping fields.

The type of a polymorphic natural join function would be

$$\forall a.\forall b.\forall c.(\text{set }[a] \times \text{set }[b] \rightarrow \text{set }[c])$$

That is, the function takes as parameters two sets and returns a third set as a result. This function cannot be written as a statically-typed polymorphic procedure, since it requires knowledge of the type structure of a, b, and c, and they are explicitly abstracted. However, the type structure is known for any particular call. Linguistic reflection allows a metafunction to be written, the generic natural join, that can interrogate the structure of the input types. At the point of use of the natural join, the metafunction is called by the compiler, and given as input the type structure. This compile-time invocation will compute the specific natural join function for these input types and a call to the specific natural join. All of this generated code is then fed into the compiler for checking and code generation.

The writing of the metafunction involves computing over the representations of the input types. Thus, the language must provide facilities for this. Indeed, it is this ability that contrasts the reflective language with other polymorphic systems. When using run-time reflection, it is also possible to examine the size and representation of the input sets and to generate an optimized algorithm.

Evolution in Strongly Typed Persistent Systems. Type-safe linguistic reflection also may be used to accommodate evolution within strongly typed persistent object stores. A characteristic of such stores is that the set of types of existing values in the store evolves independently from any one program. This means that when a program is written or generated some of the values that it may have to manipulate may not yet exist, and their types may not yet be known for inclusion in the program text.

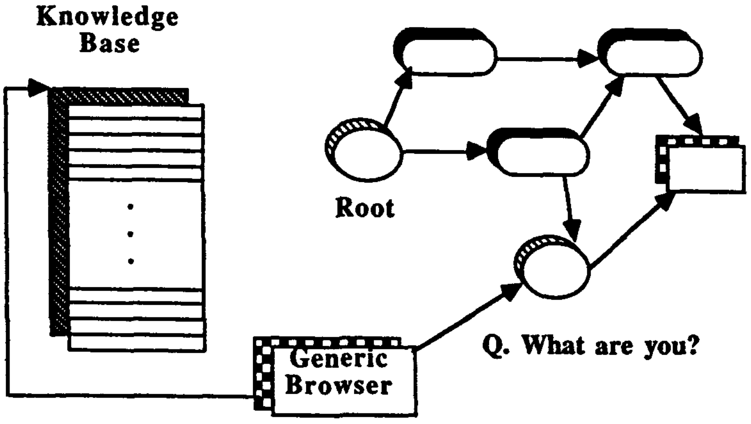
An example of such a program is a persistent object store browser (Dearle and Brown, 1988), which displays a graphical representation of any value presented to it. The browser may encounter values in the persistent store for which it does not have a static type description. This may occur, for example, for values that are added to the store after the time of definition of the browser program. For the program to be able to denote such values, they must belong to an infinite union type, such as type **any** described earlier.

However, the program cannot contain static type assertions for all the types that may be encountered as their number is unbounded. There are two possibilities for the construction of such a program: it may be written either in a lower-level technology using interpretation (Kirby and Dearle, 1990) or else using linguistic reflection.

To allow a reflective solution, the program must be able to discover dynamically the specific type of a value of the union type. Such functionality may be provided in a strongly typed language, without compromising type security, by defining representations of types within the value space of the language.

Figure 14 illustrates the use of linguistic reflection to define programs that operate over values whose type is not known in advance. The generic browser takes a specification of the object type and generates a program to browse over it. It may also store that program in the knowledge base for future use, should it encounter

Figure 14. Persistent store browser



an object of the same type again. These programs potentially perform different operations according to the type of their operands, but without endangering the type security of the system or invoking an extra layer of interpretation. The requirement for such programs is typical of an evolving system where new values and types must be incrementally created without the necessity to re-define or re-compile existing programs.

5. Achievements

The progress of persistence research can be charted by examining the proceedings of the regular workshops held by its practitioners. There are two series of international workshops: the POS (on persistent object systems) and the DBPL (on database programming languages), as well as several ad hoc events such as persistence tracks at HICSS (the Hawaii International Conference on System Sciences). These are tabulated in Table 9 for those who wish to scan this literature.

From the contents of the proceedings cited in Table 9, it is apparent that a large community of researchers are addressing persistence. This section reports some of their achievements. The persistent languages that have been developed are displayed in Table 10. A few aspects of the use of these languages are presented below.

5.1 Incremental Construction

Because PASs are long-lived, their construction usually takes place incrementally. As each requirement is recognized, or as resources become available, new suites of software and collections of data are added to the operational system. Databases have supported well the incremental addition of programs to the operational PAS

Table 9. **Major international meetings of persistence researchers**

	Date	Venue	Organizers
POS 1	Aug'85	Appin, Scotland	Atkinson, Buneman, and Morrison (1988*b*)
POS 2	Aug'87	Appin, Scotland	Atkinson and Morrison (1987)
DBPL1	Sep'87	Roscoff, France	Bancilhon and Buneman (1990)
HICSS22	Jan'89	Hawaii, USA	Atkinson and Morrison (1989)
POS3	Jan'89	Newcastle, Australia	Koch and Rosenberg (1989)
DBPL2	Jun'89	Salishan, OR, USA	Hull, Morrison, and Stemple (1989)
Security	May'90	Bremen, Germany	Rosenberg and Keedy (1990)
POS4	Sep'90	Martha's Vineyard, USA	Dearle, Shaw, and Zdonik (1990*b*)
DBPL3	Aug'91	Nafplion, Greece	Kanellakis and Schmidt (1991)
HICSS25	Jan'92	Hawaii, USA	Morrison and Atkinson (1992)
POS5	Sep'92	San Miniato (Pisa), Italy	Albano and Morrison (1992)
DBPL4	Aug–Sep '93	Manhattan, NY, USA	Beeri, Ohori, and Shasha (1993)
POS6	Sep'94	Tarascon, France	Atkinson, Benzaken, and Maier (1995)
HICSS28	Jan'95	Hawaii, USA	Rosenberg and Dearle (1995)

via their dynamic binding to schemata or views. They have made less provision for the incremental growth of schemata taking the unrealistic view that the entire schema can be designed at the outset, and that only minor modifications need be made using a schema editor. In reality, use of the schema editor is frequent, but there is little support for change in data models.

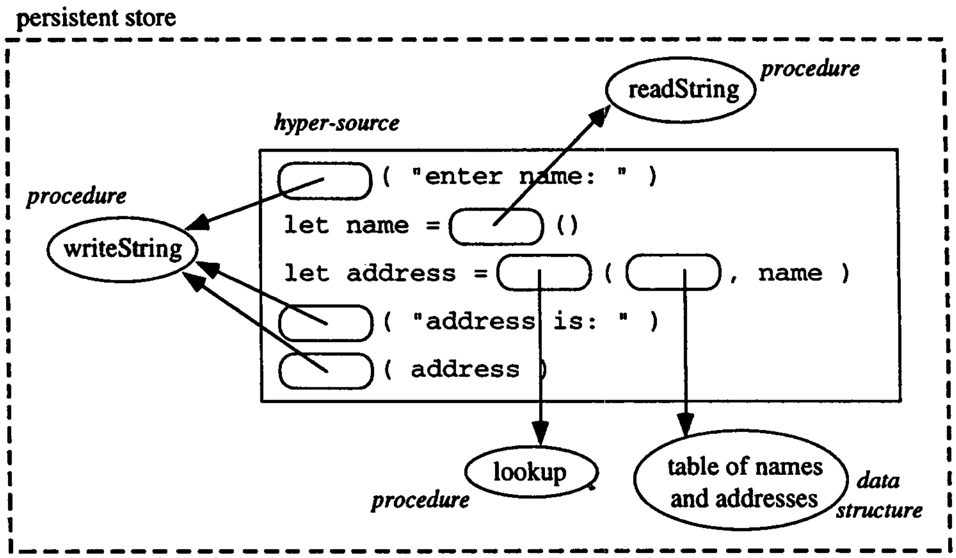
Table 10. Milestones in persistent and database programming languages

1977	Pascal/R (Schmidt, 1977)
1980	SmallTalk (Goldberg & Robson, 1983),
	PS-algol (Atkinson et al., 1983*a*),
	Plain (Wasserman et al., 1981), Taxis (Mylopoulos et al., 1980)
1981	Daplex (Shipman, 1981),
	Adaplex (Smith et al., 1981; Chan et al., 1987)
1983	Galileo (Albano et al., 1985), Modula/R (Koch et al., 1983),
	Persistent procedures (Atkinson & Morrison, 1985)
1984	Amber (Cardelli, 1986), Persistent Prolog (Bocca & Bailey, 1987)
1985	CPS-algol experiment (Krablin, 1987*b*), Poly (Matthews, 1982),
	OPAL (ServioLogic Ltd, 1987)
1986	DBPL (Matthes & Schmidt, 1989), RAPP (Hughes & Connolly, 1990)
1987	Quest (Cardelli, 1989), E (Richardson & Carey, 1989),
	χ (Hurst & Sajeev, 1990)
1988	DPS-algol (Wai, 1989), Napier88 (Morrison, et al., 1989*b*)
1991	P-Quest (Matthes et al., 1992), Staple (Davie & McNally, 1990),
	P-Galileo (Brown et al., 1992), O_2 (Deux, 1990)
1992	Hyper-programming (Kirby et al., 1992*a*), Commercial Persistent C++
1993	Tycoon (Matthes & Müßig, 1993), Fibonacci (Albano et al., 1995)
1994	Napier88 version 2 (Morrison et al., 1994*a*)

The persistence technology so far developed makes adequate provision for the incremental addition of collections of programs by providing dynamic binding constructs (Atkinson et al., 1988*a*). This is widely exploited in the prevalent persistent programming style (Dearle, 1988; Connor, 1990; Cutts, 1992; Sjøberg, 1993). Incremental addition and replacement of program parts tends to use smaller incremental units than are used with the traditional database technology. This is because the retention of types and structure and the provision of persistent bindings reduces the complexity and cost of using smaller units.

Persistence technology takes a different view over the construction of schema. It is normal to consider the design of types at the same time as the programs that will use them. Therefore, the technology itself has no bias towards programs being either more or less incremental than types. This freedom from bias appears to have pay-offs. The outline design of the data can be completed early and stored as a set of type definitions, but much of the detail can be postponed using the dynamic checking points. Then, as programs are required, they and their associated detailed

Figure 15. Hyper-program



types are designed together. This has the advantage of requiring less perturbation of the existing types, and of allowing designers and programmers to concentrate on a particular subsystem's types. To facilitate this, persistent systems have developed mechanisms for collecting types together to record and, it is hoped, to preserve this modularity. For example, Quest has modules (Cardelli, 1989) and Napier88 has declaration sets (Kirby et al., 1994*a*).

5.2 Hyper-Programming

As mentioned earlier, the presence of a persistent environment in which programs are composed, compiled, linked, and run extends the scoping of bindings to persistent values. This means that objects accessed by a program may be already available at the time that the program is composed. Where the source programs are themselves values, static R and L-value bindings may be made in the program source to persistent values. Thus, these bindings, called links in this case, can be included in the program instead of the more traditional textual descriptions of where to find persistent values. The source text of the program now contains some text and some links to persistent values and, as such, it is a non-flat representation of the program. By analogy with hyper-text, a program containing both text and links to persistent values is called a *hyper-program* (Kirby et al., 1992).

Figure 15, taken from Kirby et al. (1992), shows an example of a hyper-program. The links embedded in it are represented by non-textual tokens to allow them to be distinguished from the surrounding text. The first link is to a first-class procedure value *writeString*, which writes a prompt to the user. The program then calls another procedure *readString* to read in a name, and then finds an address corresponding to that name. This is done by calling a procedure *lookup* to look up the address in a table data structure linked into the hyper-program. The address is then written out. Note that code objects (*readString, writeString* and *lookup*) are denoted using exactly the same mechanism as data objects (the table). Note also that the object names used in this description have been associated with the objects for clarity only, and are not part of the semantics of the hyper-program.

Figure 16, again taken from Kirby et al. (1992), shows an example of the user interface that might be presented to the programmer by a hyper-program editing tool. The editor contains embedded light-buttons representing the hyper-program links; when a button is pressed the corresponding object is displayed in a browser window. The browser is also used to select persistent objects for linking into hyper-programs under construction.

The benefits of hyper-programming have been discussed (Farkas et al., 1992; Kirby, 1992*b*; Kirby et al., 1992) and include:

- the capacity to perform program checking early—access path checking and type checking for linked components may be performed during program construction;
- the ability to enforce associations from executable programs to source programs—links between source and compiled versions may be used;
- support for source representations of all procedure closures—free variables in closures may be represented by links, thus allowing hyper-programs to be used for both source and run-time representations of programs; and
- increased program succinctness—access path information, specifying how a component is located in the environment, may be elided.

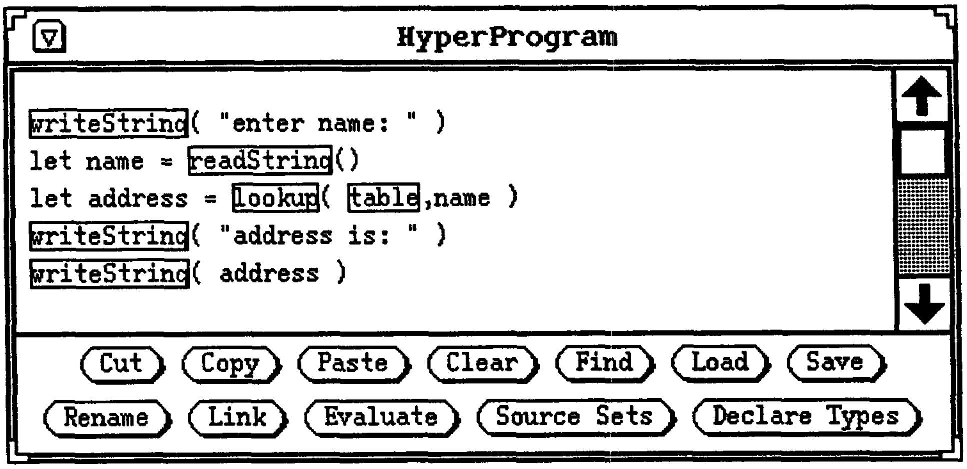
5.3 Persistent Workshop and Methodologies

Persistent programmers need virtually all the tools in a programming environment that are required by other programmers (e.g., editors, compilers, build-managers, version-managers). There are, however, some important differences in their requirements and, by exploiting persistent technology to provide this environment, their needs can be met in novel ways.

An experiment is underway in building such an environment, called the persistent workshop. The experiment has three purposes:

- to demonstrate the ease of construction and new functionality afforded by developing the workshop using orthogonal persistence;
- to provide a good programming environment for persistent programmers; and
- to set up an example of an operational PAS so that its usage and behavior may be studied.

Figure 16. User interface to a hyper-program editor



```
writeString( "enter name: " )
let name = readString()
let address = lookup( table,name )
writeString( "address is: " )
writeString( address )
```

All the source and operational code, and any design material, are held in the persistent store. All the tools communicate via the persistent store. Dynamically typed interfaces are used where independent tool evolution requires flexibility. The workshop provides a set of tools and an extensive library of components (Atkinson et al., 1993*a*; Kirby et al., 1994*a*). Examples of subtly changed requirements on these tools arising from persistence include:

- tools that provide a means of configuring a workbench, establishing a style of working, and a particular view of the persistent store—these enable programmers to limit the complexity of data and choices open to them to those relevant to their task (Waite, 1995);
- tools to examine the relationship between the persistent store and programs—these enable programmers to more quickly appreciate the existing structures and to verify the results of their work (Lavery, 1995*a*, 1995*b*);
- tools to organize and build libraries of program parts and data, and to search these libraries using information retrieval techniques—which are intended to improve component re-use both within a PAS and among PASs (Brown, 1993);
- tools to manage the large number of incremental units, including providing aids for building and installing subsystems, and to verify that the parts comply with a methodology (Sjøberg et al., 1995) and
- tools to assist with change management (Sjøberg, 1991).

These tools already exploit properties of persistence. For example, they leave data structures in the persistent store to accelerate subsequent operations. As another example, the build manager, the store analyzers, the methodology checker, and the change management aids all rely on being able to scan the store and on knowing that, unlike many systems, all external information, vital for its construction, is explicitly connected to the PAS's representation.

Two methodologies are supported in the workshop at present. One is closely allied to traditional database practice. In that, the PAS construction proceeds via the design of a system in some conventional data model (Cooper and Qin, 1994). The types and programs are then automatically derived. This works reasonably well on a small scale, but will not yet scale up, because incremental construction via automated generation is still a research issue. The other uses a methodology called SPASM (Sjøberg et al., 1995), and sets out to delimit dependencies between incremental components accurately. This involves the choice of particular ways of using the persistent technology, but has the advantage that more of the build and change processes can be automated.

5.4 Persistent Software Engineering

The concept of the hyper-program, containing links to persistent objects and made possible by the provision of strong typing and referential integrity, has a number of applications in software engineering environments. Three examples of such usage are: the Napier88 system, the Flex system (Currie, 1985; Stanley, 1986; Stanley and Drummond, 1988) and the Vesta Configuration Management System (Chiu and Levin, 1993; Levin and McJones, 1993).

The Napier88 system contains a fully functional hyper-programming system and, in addition, contains prototype versions of the following further uses of hyper-links in the software engineering context (Morrison, et al., 1995):

- simplification of the programming model via hyper-code;
- version control;
- configuration management; and
- documentation.

One of the advantages of hyper-programming is the ability to use the hyper-program representation for both source and run-time representations of programs. At program composition time, the programmer may construct a hyper-program using a tool that is a combination of an editor and a browser. At run-time, the hyper-program representation also may be used to represent an active computation. This is possible due to the non-flat nature of that representation. Free values (i.e., non-local references to objects and procedures) may be represented as links and the inherent sharing of values and locations referred to by links is preserved. This is not possible with textual representations of programs since the sharing is lost.

The hyper-code abstraction allows a single program representation, the hyper-program, to be presented to the programmer at all stages of the software development process. In constructing a program, the programmer writes hyper-code. During execution, during debugging, when a run time error occurs or when browsing existing programs, the programmer is presented with, and only sees, the hyper-code representation. Thus, the programmer need never know about those entities that the system may support for reasons of efficiency, such as object code, executable code, compilers, and linkers. These are maintained and used by the underlying system but are merely artifacts of how the program is stored and executed and, as such, are completely hidden from the programmer. Only one set of tools is required for manipulating the hyper-code, thereby simplifying the user interface and the complexity of the system implementation. This permits concentration on the inherent complexity of the application, rather than on that of the support system.

Software environments are made safer but less flexible whenever a name binding can be replaced by a hyper-link. The replacement of name bindings found in traditional software systems by links and reverse links ensures that a given software component and the objects to which it links remain accessible from one another, as a consequence of persistence defined by reachability. Thus, in version control, configuration management, and documentation systems, the software items can refer directly to the components to which they relate. Program and documentation may be linked directly together with hyper-links. A particular configuration may have a hyper-link to a configuration script, and a version may have direct links to related versions.

By using similar methods, the same advantages accrue to other activities supported by software environments which are not discussed here, such as debugging, profiling, and optimization (Cutts, 1992).

The Flex system (Currie, 1985; Stanley, 1986; Stanley and Drummond, 1988) consists of a programming language supported by a persistent file store that contains structured data. Since the Flex language is a complete version of algol-68, it has all the ingredients required for the exploitation of strongly typed persistent linkage, namely: persistent links (capabilities in this case), higher-order functions, and strong typing. Flex also has the concept of hyper-links, called cartouches after their user interface representation. Source code may contain cartouches which point to persistent objects, in this case typed files. The notion of pairing source code with executable code so that they only can be updated in lock-step is also present in Flex.

The Vesta configuration management system (Chiu and Levin, 1993; Levin and McJones, 1993) is based on a system that does not allow overwriting in place. Since the system is Unix based, inodes are unique. Configurations are guaranteed to refer to a new file if the source has changed. To support this, the implementors re-wrote the Unix system to prevent update in place and thereby support configuration management by persistent links with referential integrity. Build avoidance was achieved by caching function applications. These identities were essential for recognizing re-evaluation with the same argument.

6. Future Directions

An overview of the current state of persistent system research has been presented. There are commercial applications (Greenwood et al., 1992) and many experimental ones (Grossman et al., 1994; Reinwald et al., 1994). These indicate the promise of orthogonally persistent object systems, and that the technology is already usable. However, much remains to be done to achieve the full potential of persistence, and to establish this approach as one that can be safely adopted by industry. These further avenues of research and development are presented under three headings:

- *Extensions of Persistence.* Developments that will make the facilities integrated within orthogonally persistent systems more complete;
- *Exploitation of Persistence.* Persistent methodologies, evaluation, and use of the existing persistent technology; and
- *Delivering Persistence.* Continuing improvement to the engineering of persistent support systems.

6.1 Extension of Persistence

Extensions of persistence fall into two categories: those which are necessary to facilitate PAS design, construction, maintenance, and operation already well established in the database context, and those which meet new needs currently unmet by any technology.

To meet the first requirement, which is here called "database completeness," the facilities for data modeling, bulk types, transactions, distribution, autonomy, and evolution all deserve further attention. The second requirement overlaps with the first, in that better data models informed by research into types, more varieties of transactions, complex transactions that integrate well with recovery, etc., are required in both contexts. New extensions in the second category include seeking the conceptual simplicity of persistence in the combination of programming languages with other sub-systems, such as: user-interface management systems, operating systems, and communication systems.

6.1.1 Conceptual Modeling and Generic Interfaces. The use of types to describe data was illustrated earlier in this paper. Some would argue, however, that these type notations are insufficiently descriptive, or fail to exploit the body of knowledge regarding database design using, for example, E-R models, relational models, semantic models, or object modeling. There always will remain application domains where a new group of data modeling constructs will be useful. The utility of providing extensible libraries of type constructors requires validation in a realistic PAS development environment.

New type constructors can be defined and *named* using the pre-existing or pre-defined constructors. Cooper has extensively explored the process of mapping traditional data models to types in several different languages, and has used reflection to build aids to performing this mapping automatically (Cooper, 1990*a*; Cooper and

Qin, 1992; Cooper and Qin, 1994). His systems automatically generate all the types that match some data model and both a text-based and diagram-based schema editor. They also generate the usual interface components (e.g., forms generators), skeleton application programs, and a relevant library of persistent components. These should aid the transfer of existing system design skills to the new technology, migration of designs, rapid prototyping, and application construction. That expectation needs to be validated via real applications.

Others have developed more specific mappings from models to persistent type systems (Stemple et al., 1992*a*; Sheard and Hook, 1994; Wetzel, 1994; Albano et al., 1995). It is likely that this approach of mapping to type systems using reflection will prove widely applicable, and will relieve the pressure on type systems to develop to the levels of complexity where they are difficult to understand and expensive to implement.

It remains to be seen whether the code and constructs to accommodate changes specified in terms of such higher-level models can be generated automatically and applied. If this is achieved, then the provision of high levels of data independence can be automated for a wide range of modeling styles (Atkinson et al., 1993*b*).

It is probable that there are considerable benefits to be gained by generating code from many high-level notations into a common orthogonally persistent system. The composite would have consistent behavior under stress (e.g., recovery). It is expected that the use of reflective generation will allow such a persistent target language to be simplified. The authors do not know of any exploration of this potential.

6.1.2 Bulk Types, Query Languages, and Optimization. An important component of data models is bulk types (Atkinson and Buneman, 1987; Atkinson et al., 1991). Their value is three-fold:

- they abstract over the size of collections they represent;
- they allow the description of important regularities in the data; and
- they permit exploitation of the regularity (e.g., in query notations and in the optimization of operations and data movement.

Recent work shows that certain classes of bulk types can be treated consistently (Ohori et al., 1989; Trinder and Wadler, 1989; Breazu-Tannen et al., 1991; Buneman et al., 1994). It has also been shown that these bulk types can be implemented by mapping them to the underlying system (Cooper and Qin, 1992; Ghelli et al., 1992; Matthes, 1992). If this target system has orthogonal persistence, then the bulk values inherit that persistence. It has long been argued that, during this mapping, the traditional database optimizations can be accomplished (Cooper et al., 1987). These optimization techniques have yet to be convincingly demonstrated, though it still seems a valid proposition. A potential problem is whether the "standard" primitives are the appropriate target, and whether their cost models are well enough developed to support optimization.

The issue of whether to build-in or add-on bulk types is often posed (Matthes and Schmidt, 1991). The resolution is unlikely to be simple. The addition of new bulk-types will always be required, as it is inconceivable that all potential requirements could be anticipated. However, their definition in terms of primitives will remain a skilled craft. In an attempt to de-skill it to some extent, and to provide a more appropriate target onto which all bulk-types could be mapped, a generic primitive has been suggested (Atkinson et al., 1991).

Languages continue to be designed with built-in bulk types (e.g., Fibonacci). One reason for this is the issue of query notations. An alternative approach that is worth further exploration, is to support appropriate notations by using reflection to generate code from a suitable query notation.

Query languages and bulk types are intimately interrelated in the context of object models and persistent type systems. Queries can only be effectively expressed over collections of values or objects, which may be represented by bulk types. But aspects of the query (e.g., selection expressions) may require expressions based on any operation of any other type in the language, including other bulk types.

6.1.3 Transactions and Recovery. Typical persistent systems implement recovery well. Similarly, typical database systems implement recovery effectively in the context of serializable transactions. However, both show limitations when any more sophisticated form of transaction or concurrency is required.

Research is needed to validate the primitives and constructors that allow a range of transaction types and concurrency models to be used with the same system (Krablin, 1987*b*; Stemple and Morrison, 1992; Munro et al., 1994). Again, generation and translation technology is anticipated to prevent the target becoming over-complicated, and to relieve programmers from mechanizable detail. It remains a challenge to develop a recovery mechanism that has comprehensible behavior in the context of the full range of transactions.

6.1.4 Evolutionary Constructs and Mechanisms. Infinite union or dynamic types, such as type **any**, provide the basic requirements to permit incremental changes to types. Reflection permits software to change its environment, and to accommodate change in ways that are otherwise impossible without recourse to expensive interpretation of all operations. Exploration of how these may be combined to support system evolution has already made some progress.

The mechanisms for changing program parts are well developed in orthogonally persistent systems (Connor, 1990; Cutts, 1992; Sjøberg, 1993), as the executable code values can be transactionally replaced via store updates (Connor et al., 1994*b*). There are, however, many challenges that remain.

- *To provide configuration and version managers.* The challenges here are, in part, more complex than with traditional technology, since all types of data and associated types and programs need to be consistently versioned or configured. However, persistent technology may also come to our aid here. Its provision for all types of data, and its support for reliable references, as well as its programming benefits combine to assist in the construction of these tools (Morrison et al., 1994; Morrison et al., 1995).

- *To provide convenient notations for specifying change.* One line of attack is to schema edit the higher-level data and process models, and to generate the underlying changes. This runs into difficulties, as programmers have often provided extra information during these mappings, which it is unfeasible to expect them to re-supply for each change. Another strategy is to support type editing on the types defined in the orthogonally persistent system itself. This pre-supposes that the mappings are comprehensible or it abandons generation from high-level notations.
- *To provide efficient propagation of change to the existing instances of types.* One strategy that works is to use a stable anchor for each type and accommodate extensions. This is very similar to the method commonly used for implementing object specializations as fragments. However, it has costs of indirection and type projections. It can, in principle, be automated apart from generating new values, and can be operated either in batch or incremental mode (Clamen, 1994). But the trade-offs and implementation have yet to be explored in this context.
- *To assist programmers in managing change.* This requires recording or discovering dependencies, performing impact analysis, and identifying the localities that need to be changed. Orthogonally persistent systems often hold the source code within the same regime as the types, data, and executable code. Therefore, it is possible to search for these dependencies, and to present them (as demonstrated by Sjøberg, 1993). The development of change management tools and methodologies for organizing persistent object stores so that handling the consequences of change is tractable, is still in its infancy.
- *To provide appropriate change absorbers.* View mechanisms in databases act as change absorbers. That is, they are (usually manually) revised to prevent the propagation of change, and the explicit use of a view also identifies very rapidly the cases where a program cannot possibly be affected by a schema change. The parsimonious use of explicit bindings between programs and stores, as typified by Napier88, also provides the rapid elimination of the unaffected programs, and abstract data types can, in principle, provide manually maintained change absorbers (Connor et al., 1990*b*). Automated change absorption should be possible in many cases, and the relevant methods and mechanisms need development before they can be used for large scale applications (Atkinson et al., 1993).

It would be unwise for any enterprise to commit seriously to a technology purporting to support persistence if they were not confident that it adequately could support schema change. Where persistent systems have imprecise type knowledge (e.g., those based on C and C++), it is doubtful whether there is enough information to safely support change.

6.1.5 Distribution and Autonomy. Persistent systems abstract over the locality of data, that is, the programmer does not need to be aware of whether it is on disk or in RAM. One approach to distributed orthogonally persistent systems is to continue this location transparency (Wai, 1989; Koch et al., 1990; Gruber et al., 1992; Daynès and Gruber, 1994; Dearle et al., 1994). This abstraction eventually may satisfy those that use distribution to achieve reliability and performance through distributing computation and migrating computation or data, but it depends on the discovery of adequate algorithms for automating data and program movement. Although this approach offers desirable semantic simplicity, it does not scale indefinitely, fails to accommodate change, and does not have a realistic approach to failures (Mira da Silva et al., 1995).

Others, however, wish to exploit their knowledge about processes and the network to write efficient distributed persistent systems. This requires that the locality of data be exposed (Liskov, 1988; Liskov et al., 1990; Munro, 1993; Mira da Silva, 1995). As yet, no tidy resolution of these conflicting requirements is in sight, and different researchers are pursuing the two models. *Obliq* is an interesting compromise (Cardelli, 1995), as it allows its programmers to indicate whether data or method should migrate but does not explicitly specify locations.

Generally, providers of PASs have to provide high levels of availability. However, distributed systems are prone to various forms of failure. It appears to be necessary to give programmers access to failure information so that they may program this availability and inform users about the causes of failures and recovery expectations since the appropriate action is highly application dependent. But, to provide this information, the persistence abstraction has to be compromised. There is a challenge here: how to present to programmers a model of distribution that includes autonomy and failure, but does not leave them to handle all problems unaided.

An important reason for distribution is ownership. Different parts of the system belong to independent organizations who retain the right to make changes as they see fit. This autonomy is, in fact, independent of distribution, and mechanisms that support it also contribute to scalability and the accommodation of change (Atkinson et al., 1993*b*).

The crucial input that persistence brings to the more general search for adequate models of distribution and autonomy is a commitment to retaining simple, combinable primitives, even if this raises additional implementation challenges.

6.1.6 Scalable Systems. Over the lifetime of a PAS, it is common to experience both massive growth and contraction, both of which may be difficult to predict. Often these variations in scale affect only parts of the PAS; for example, meta-data, data, program, and users all accumulate around successful parts of a PAS while other parts may virtually atrophy. Technology is required that adapts well across all possible sizes of component collection. Although the range that can be efficiently accommodated is growing, and specific solutions have been found for very large bodies of data (Grossman et al., 1994), there is always a need for better adaptive algorithms.

However, there are limits to the performance that can be expected from algorithms that manage data on disk. For example, disk garbage collection algorithms and store re-organization algorithms will never be feasible for arbitrarily large bodies of unconstrainably interconnected data. One suggested approach is to introduce logical partitions that assist with scalability throughout the life-cycle. A complementary strategy is to utilize replication and parallelism in distributed systems.

Persistence researchers have a particularly difficult time verifying that their approaches to scale and evolution will actually work. It is very difficult to resource experiments of the required scale, realism, and duration.

6.1.7 Integration With Other Domains. Much of the safety and simplicity of orthogonally persistent systems is achieved by postulating a closed world (e.g., the well-defined universe of discourse determined by the type system, or the well defined extent of meta-data, data, and program determined by reachability). These totally closed worlds are unrealistic if transition is to be made to persistence. Indeed, at present, most persistent systems have connections with the external world (for example they can use UNIX files and UNIX shell commands).

More sophisticated interfaces obviously are needed. It has been remarked that the work on mapping data models already supports some skill transfer. It also could be used to automatically generate bulk load and unload programs that are still type safe, for any of the data models. This could be extended to use the selective loading developed by Abiteboul et al. (1993).

A data type complete RPC technology that uses the automatic generation of interfaces via reflection has been developed (Mira da Silva, 1995), but there are still problems to be overcome (Mira da Silva et al., 1995). This kind of RPC can be extended to automatic translation at the interface and connection to processes in other languages. Indeed, as long ago as 1982, Buneman demonstrated how type-safe interfaces could be generated to external data collections (Buneman et al., 1982). More recently, Matthes has repeated that work in the context of a persistent system (Matthes, 1992). Current persistent type systems permit these external data repositories to be safely modeled as abstract data types. Reflection allows these abstract data types to be automatically generated when they are needed, avoiding the high set up costs and the large name spaces of Buneman's mechanism.

Similarly, standard interfaces to a persistent system, or to an external system can, in principle, be generated. The extensive use of such generated, safe interfaces has not yet been investigated. It is not expected to raise conceptual difficulties beyond those raised by autonomy and distribution.

6.2 Exploiting Persistence

There are several ways in which orthogonally persistent system research may be exploited. Methodologies and tools can be developed that enhance the utility of the existing and envisaged systems. The concepts developed in one case can be re-applied elsewhere; in particular, programming languages and OODBMS can be modified to utilize the results.

6.2.1 Building and Observing Persistent Exemplars. The developers of orthogonally persistent systems believe that these systems are ready for serious evaluation. Wider use is needed to test their utility, and to provide feedback to implementors so that the engineering issues can be better understood. Many of the issues are apparent only when the usage involves large volumes of program and data. It is crucial for effective evaluation and the generation of useful characterizations of load, that the usage of these systems is realistic, in the sense that operational loads, data sets, programmers, users, and changes are typical of real applications. Therefore, experiments should have sufficient resources to explore the critical issues at adequate scale and over sufficient elapsed time (Atkinson, 1992).

Good quality engineering depends on the systematic use of measurements. Therefore, the persistent community needs to develop system models and measurement technology and practices so that PAS loads and the corresponding performance of PASs can be accurately described and compared, and so that design trade-offs can be based on reliable information (Atkinson et al., 1992; Scheuerl et al., 1995).

6.2.2 Using Type-safe Linguistic Reflection. Reflection is especially significant in persistent systems, because the program generation costs can be amortized over more applications using persistent memoization, and because it is the basis for system evolution. It has two important roles: it allows adaptive behavior to be programmed, and it facilitates the writing of processors from specialized notations to a target persistent language. Both techniques have already been well demonstrated. Two lines of research are identifiable.

- Research to improve the notations and libraries available for those writing reflexive programs is already underway (Cooper and Kirby, 1994; Kirby et al., 1994*b*). Further elimination of the noise generated by embedding representations of a program within programs is probably relatively easily achieved. The more challenging task is to help the programmer think simultaneously and accurately about two computations: the computation to form the new code and the computation that code will eventually perform. Part of that challenge may be to make more apparent the way the environments of the two computations relate.
- Research also is needed to explore the full potential of reflection. Examples that have been mentioned already are the construction of optimized strategies for computations over bulk values based on observed properties of those values and the simplification of target persistent languages given the ready availability of reflection. The research into the provision of data models and user interface generators is well underway. Research into the use of reflection for generating safe external interfaces will continue. The use of reflection to automate the management of the consequences of change is a new area

ripe for exploration. These are just a few of the possible applications of reflection deserving exploration.

Given the advent of type-safe reflection and tools that assist programmers to use it, there is likely to be an explosion of applications. While these may be important in themselves, it is perhaps even more important to extract and identify effective methodologies for reflection to enable other programmers to use it well.

6.2.3 Methodologies and Tools. Constructing application systems using orthogonal persistence allows new techniques and new structures. For example, because structural information is neither lost nor obscured if it is left in the store by one program to be used by another, there is a trend towards partitioning applications into smaller units. As another example, libraries and application programs both can be built incrementally in the persistent store using the same transactional mechanisms as the application uses. This permits incremental construction and replacement of small units of program and data, even while the system is in use.

Because experience is gained with writing PASs, methodologies appropriate for this new technology are emerging (Dearle, 1989; Connor, 1990; Sjøberg, 1993). These require further development and need to be made accessible to application programmers in a variety of persistent technologies including OODBMS through the provision of tutorials, tools, and exemplars.

Crucial to making these methodologies usable is the provision of an appropriate persistent programming environment. Three experiments are underway in this direction at present: the hyper-programming workbench (Connor et al., 1994*a*), the PIPE programming environment (Dearle et al., 1992*a*), and the persistent workshop (Sjøberg et al., 1995). The hyper-programming workbench is allowing experiments with new construction techniques and with construction-time type-checking. The persistent workshop already provides a set of tools to help programmers, including:

- specialized editors, automated binding resolution, and compilers;
- library management tools, including aids to finding components based on information retrieval techniques; and
- visualizers that are intended to help programmers understand the state of the persistent store.

These tools will continue to develop as programming aids. The tools that derive dependencies, and automate re-compilations and re-executions that are under development, lie at the boundary between supporting individual programming and supporting software engineering. If a methodology has been accurately followed, these tools can be more precise. Consequently, the facilities for verifying consistency with a methodology are under development (Sjøberg et al., 1994*a*). The methodology has been chosen in the belief that it will facilitate the change processes that dominate a PAS life-cycle. This has yet to be verified, and even the advice to the programmers on the consequences of change is still in its infancy (Sjøberg et al., 1993; 1994*b*).

6.2.4 Influencing Other Technologies. The results achieved by applying the persistent design principles can be applied in other domains. Three such areas where persistence results could have a beneficial influence are: programming languages, OODBMS, and operating systems.

Existing commercial programming languages (e.g., C, C++, Ada) commonly are used in building PASs. There is a natural desire to produce persistent versions. The persistence results show that this can be done safely only if the types of each value are known unambiguously. Therefore, it is probable that it will be worthwhile to establish persistent dialects of these languages *that have this safety property.*

From time to time, new programming languages emerge. The designers of those languages will have omitted important and feasible facilities if they fail to include orthogonal persistence.

Object-oriented databases are following a path similar to that followed by persistent technology. The persistent experience can guide OODBMS design and implementation. For example, basing persistence on reachability from identified persistent roots and providing orthogonal persistence now appears obvious, and has been adopted in O_2 (Bancilhon et al., 1992). OODBMS also would benefit from properly incorporating the stored program (methods) into the persistent store, so that consistent bindings between code and data can be maintained (Morrison et al., 1995).

Operating systems already provide non-orthogonal (weakly-typed) persistence in the form of file stores. It is apparent from the complexities of using their environments (e.g., **setenv** in UNIX) that orthogonal persistence and better defined models of binding to the persistent store would be a significant advantage. Some work also is underway on persistent operating systems with a view to having such facilities and supporting persistent programming generally in a much more efficient way. Certainly, the requirements of orthogonal persistence place new demands on the low-level mechanisms provided by operating systems. Persistence research, therefore, should influence the way operating systems present their storage facilities and the facilities they offer to system implementors. But persistence also may have another role to play, because the harmonization between databases and programming languages (which has been its main focus to date) also could be applied to the relationship between programs and the operating systems they use. Why not have the same naming, binding, and typing rules when using operating system facilities (files, system calls) as are used in the language? Of course, as was the case with databases, both ultimately would evolve to meet their common requirements.

6.3 Delivering Persistence

Although reasonable scale and performance can be obtained from current orthogonally persistent technology, there is a continuing search for better engineering. The challenge is to develop persistent object stores that provide the full range of facilities for collections of data from a few thousand bytes to terabytes.

Four models of providing persistence have been presented.

- *Provision of a "library" of persistent facilities in a standard language.* This may be a useful pragmatic intermediate step, but we believe it is unlikely to lead to safe and comprehensible PAS building facilities.
- *Combination of a standard language and a separate store.* This is commercially a most active line of attack at present, because various OODBMS vendors combine their product with various languages. In reality, they often modify the implementation of the language, and adapt the provisions of their store. This is, therefore, a valid intermediary process. It may be difficult for it to give satisfactory results in the long term, due to the lack of reliable type information and inconsistent behavior when the system is stressed. Certainly, the potential of this approach to support PAS evolution should be investigated *before* it becomes commercially critical.
- *Integrated design and provision.* This approach has been taken by the persistent programming community. It is clearly a radical and, therefore, risky departure from commercial practice. However, this technology is at least as mature as relational technology was when it was first explored commercially.
- *Rebuilding the whole support system.* This approach starts with the construction of a persistent operating system (or even hardware) and is, therefore, a much longer term program of research. Without its results, persistent support systems will continue to conflict with operating systems at a cost of performance and, possibly, of reliability. This avenue of research is, therefore, important, because it will develop technology that will be required as persistent systems become prevalent.

Only by building and using operational quality systems that take one of the last two forms above can the real potential of orthogonally persistent systems be verified (Atkinson, 1992).

6.3.1 Efficient Object Stores. Three strategies are in wide use to provide persistent object stores:

- Constructing above-standard operating systems using block transfers between files. This has the advantage of portability.
- Constructing above-standard operating systems using memory mapping technology. This exploits paging hardware, but has less locality, and may suffer from slow operating system interfaces and inappropriate protection granularity. It can have the advantage of running programs written in a non-persistent language in a persistent mode (Singhal et al., 1992).
- Constructing above-light-weight or specialized operating systems using memory mapping. In some cases these experiments also exploit specialized hardware. Very little portability is available, and performance gains are as yet unproven.

Orthogonal to the above class of design decisions is the extent to which the long-term form of an object differs from its active-computational form. For example, pointers can be radically different (Koch et al., 1990; Moss, 1990; Suzuki et al., 1994, Kemper and Kossmann, 1995), code can be target independent (Atkinson et al., 1993*b*) and data may be compressed and encrypted.

There are many other variations in detail that are currently being explored, for example, the mechanisms for recovery (Scheuerl et al., 1995). It is clear that the space of alternative store designs is very large and is, as yet, poorly charted. Research is needed to develop reliable information to guide store implementors, and to provide high-performance stores with predictable behavior over a range of loads.

The situation is much more complex when forms of distributed persistent store are attempted. There are some experiments in client-server technology (Dearle et al., 1992*b*, 1994) but there is, as far as the authors know, little work yet on generally distributed stores on a cluster of well-connected machines.

6.3.2 Efficient Bulk Types. The requirements for bulk types are well established (Atkinson and Buneman, 1987), and their implementation has received much attention from the database community. Their key achievements should be transferable to the persistent context. However, this may result in pressure to change aspects of the store implementation and even language primitives. Some compromise between totally add-on and totally built-in provision is likely to prevail eventually. Full scale optimization cannot be implemented without better mathematical models of store behavior. These need to be vigorously sought and experimentally validated.

6.3.3 Code Generation for Reflexive Higher-Order Persistent Languages. There are several difficulties with code generation for higher-order languages. For example, it is convenient to use some intermediate target language such as C to avoid computer architecture specificity. This requires techniques for handling higher-order procedures such as those in Appel (1992). But orthogonal persistence requires that the generated code reside in the transactional stable store. Consequently code (or more precisely, procedure closures) must be represented as objects that can be shipped in and out of store, garbage collected, and stabilized (Atkinson and Morrison, 1985). Furthermore, these procedures may be generated during execution as a consequence of run-time reflection. Once distribution or longer-term persistence is required, these procedures must be loadable onto any architecture. This combination of requirements is challenging. Consequently, only prototype code generators exist at present that meet all these requirements (Bushell et al., 1994).

6.3.4 Pre-populated Stores. A persistent store can be shipped in pre-populated form, as are many object-oriented languages and OODBMS (a good example is the SmallTalk persistent virtual image). The challenge is to determine what should go into this initial population and also to find ways of organizing it and accessing it that enable the majority of programmers to make good use of its facilities. Not only will this pre-population contain an extensive library of useful code, but it also can contain useful data structures (e.g., tables of unit conversion constants, cartographic images, pictures). Furthermore, the range of code that can be usefully shipped includes access algorithms (e.g., $B+$-trees, R^*-trees), which can be already tuned to match the properties of the shipped store. The development and consideration of these issues can be conducted by a wide community of researchers and eventually we should have the PAS equivalent of good numerical algorithms libraries with much more convenient mechanisms for using them.

Typically, new releases of these store populations are built, and users who are already using an earlier version may want to preserve their work but also avail themselves of the new libraries. This is just one example of a collection of problems concerned with merging, replacing, and shipping subsets of the objects in one store to other stores.

7. Summary and Conclusions

An overview of orthogonally persistent systems has been presented, starting from their motivation and design principles. These achieve simplicity from consistency and regularity. The lack of exceptions enhances the power, while the reduction of issues to be dealt with by programmers reduces application design, construction, and maintenance costs.

Languages and systems that achieve orthogonal persistence have been reported. Crucial to their achievement of integrating database and programming requirements is the recognition of the close relationship between schema and types. The challenge of obtaining the correct balance between precision and description is addressed by the introduction of type operators, and the concomitant use of polymorphism. This allows adaptable type systems to be developed with power similar to existing data models, and an extensibility necessary for long-term service (Connor et al., 1991; Connor and Morrison, 1992).

The introduction of dynamic checking points within the type system serves three purposes: it allows type checking to be incremental and, therefore, feasible in large applications; it allows incremental binding; and it permits localization of schema changes to subgraphs delimited by these points. The support for incremental evolution of persistent applications requires new binding mechanisms that utilize the dynamic checking, and provide consistent naming schemes throughout applications.

The development of constructs that provide concurrency, transactions, and recovery taken together still lacks unifying concepts that cover the full range of possibilities from those present and sought in databases to those available in programming languages. However, solutions exist for important subdivisions of that space and the present lines of research are rapidly expanding these.

Much research has focused on how to support orthogonally persistent systems. Four architectures are all being explored. The least committing involves extending an existing system with libraries and possibly pre-processors. Although this can provide short term benefits with few risks, it has fundamental limitations in the longer term. A common strategy is to combine two existing systems. This has operated since the early days of databases, but the degree of integration has advanced considerably with the advent of OODBMSs. It may still leave programmers coping with two models, particularly when they are concerned with failure semantics and other symptoms of system stress. They also may find that they still have to perform mappings between representations, though this is now much diminished.

Recently, integrated systems have been constructed in standard computing environments. These achieve the desired consistency and are portable. They are, at the very least, a necessary intermediary on the path to wider use of orthogonally persistent systems, and are likely to have long-term utility. Their only drawback is that they operate in an environment tuned to a different style of computation and, therefore, may have efficiency problems due to this mismatch. Although this may present challenges to current implementors, it is feasible that the supporting operating systems and hardware can evolve to meet the new needs of this architecture.

The final architecture may be viewed as prototyping this evolution or as a platform for a more radical transition. In this architecture, the underlying operating systems and even the hardware are re-designed and re-implemented to provide orthogonal persistence for all the systems they support. This radical replacement of the persistent support system would, if it were eventually adopted, have the benefit of ensuring consistent persistent behavior, even under stress, of all the software and data it supported, simplifying the tasks of programmers and users.

In all of the last three avenues of research, much effort goes into object store design. The challenge is to find mechanisms for object movement, representation, translation, concurrency, transactions, recovery, and space management that combine well together on the available hardware and software platforms. The size and load on parts of a persistent system is likely to vary dramatically between applications, and during the lifetimes of some applications. Consequently, there is considerable interest in adaptive algorithms. Persistent object store technology is now quite sophisticated, and can deliver reasonable performance and full functionality over specific areas of this design space.

Type-safe linguistic reflection has been developed extensively in the context of persistent systems for three reasons: it enables systems to evolve; it allows specialized notations to be used, from which equivalent types and programs are generated in the persistent language; and it permits safe computations over types yielding greater genericity.

Using these technologies, substantial applications have been built. Their construction and maintenance uses incremental techniques that originated with database programming but finer grain and more general incrementality now is possible. Among the applications are tools and programming environments that support persistent programming and exploit the potential of persistence to give reliable access to contextual information. Hyper-programming is supported in such an environment. This is a new technique for program construction that forms static bindings from the source program representation to values in the store while the source is being edited.

The present uses of persistence for constructing programming environments and software engineering tools demonstrate not only the viability of persistent technology but also its ability to support the introduction of new tools, flexibly coupled via a store that preserves all structural information and affords continuous type safety. The software engineering tools particularly exploit the reliable references and the data modeling tools make substantial use of reflection.

Many avenues of research are being pursued in persistent systems, and many more are opening up. The three main avenues of research are: the extension of persistence results to other contexts; the exploitation of the potential of persistence to facilitate new applications with sophisticated computational requirements; and the search for improved engineering for persistent support systems.

After fifteen years of research, orthogonally persistent systems have reached the point where they should be used by a wider community. Researchers who build applications with complex long-term data may accelerate their experiments by using one of the more mature persistent systems. Designers of programming languages, operating systems, and database systems should at least examine the results obtained with orthogonal persistence with a view to incorporating the best and most relevant of them into their own designs. Ideally, one of the mature research systems will be commercially supported so that this new technology can be evaluated seriously under realistic use.

Acknowledgements

The authors acknowledge the support of the European Union through FIDE$_2$ (ESPRIT Basic Research Action 6309), and of the British EPSRC through grants: EPSRC Grant GR/J67611 and DRASTIC EPSRC Grant GR/J99285. The authors are very grateful for the generous help in preparing the paper given by Graham Kirby, Miguel Mira da Silva, Quintin Cutts, Paul Philbrow, and Ray Welland. The following also made helpful comments on this paper: Peter Bailey, Richard Connor, Richard Cooper, Peter Dickman, Huw Evans, Stewart Macneill, Dave Munro, Susan Spence, and Cathy Waite.

References

Abiteboul, S., Cluet, S., and Milo, T. Querying and updating the file. *Proceedings of the Nineteenth International Conference on Very Large Data Bases,* Dublin, Ireland, 1993.

Agrawal, R. and De Witt, D. Recovery architectures for multiprocessor database machines. *Proceedings of the SIGMOD International Conference on Management of Data,* Austin, TX, 1985.

Akima, N. and Ooi, F. Industrializing software development: A Japanese approach. *IEEE Software,* 6(2):13-21, 1989.

Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. An object data model with roles. *Proceedings of the Nineteenth International Conference on Very Large Data Bases,* Dublin, Ireland, 1993.

Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. An introduction to the database programming language Fibonacci. *VLDB Journal,* 4(3):403-444, 1995.

Albano, A., Cardelli, L., and Orsini, R. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems,* 10(2):230-260, 1985.

Albano, A., Dearle, A., Ghelli, G., Marlin, C.D., Morrison, R., Orsini, R., and Stemple, D. A framework for comparing type systems for database programming languages. In: Hull, R., Morrison, R., and Stemple, D., eds., *Database Programming Languages,* San Francisco, CA: Morgan Kaufmann, 1989, pp. 170-178.

Albano, A. and Morrison, R., eds. *Persistent Object Systems: Implementation and Use.* Berlin: Springer-Verlag, 1992.

Appel, A.W. *Compiling with Continuations.* Cambridge, UK: Cambridge University Press, 1992.

Atkinson, M.P. Programming languages and databases. *Proceedings of the Fourth IEEE International Conference on Very Large Databases,* Berlin, 1978.

Atkinson, M.P. Persistent foundations for scalable multi-paradigmal systems. *Proceedings of the International Workshop on Distributed Object Management,* Edmonton, Canada, 1992.

Atkinson, M.P., Bailey, P., Christie, D., Cropper, K., and Philbrow, P. Towards bulk type libraries for Napier88. ESPRIT BRA Project 6309 FIDE$_2$ Technical Report FIDE/93/78, 1993*a*.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., and Morrison, R. An approach to persistent programming. *Computer Journal,* 26(4):360-365, 1983*a*.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P., and Morrison, R. Progress with persistent programming. In: Stocker, P.M., Atkinson, M.P., and Gray, P.M., eds. *Database, Role and Structure,* Cambridge, UK: Cambridge University Press, 1984, pp. 245-310.

Atkinson, M.P., Benzaken, V., and Maier, D. Persistent object systems. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1995.

Atkinson, M.P., Birnie, A., Jackson, N., and Philbrow, P.C. Measuring persistent object systems. In: Albano, A. and Morrison, R., eds. *Persistent Object Systems,* Berlin: Springer-Verlag, 1992, pp 63-85.

Atkinson, M.P. and Buneman, O.P. Types and persistence in database programming languages. *ACM Computing Surveys,* 19(2):105-190, 1987.

Atkinson, M.P., Buneman, O.P., and Morrison, R. Proceedings of the persistence and data types workshop, Appin 85. Universities of Glasgow and St Andrews Technical Report PPRR-16-85, 1985.

Atkinson, M.P., Buneman, O.P., and Morrison, R. Persistent object systems: Their design, implementation and use, Appin 87. Universities of Glasgow and St Andrews Technical Report PPRR-44-87, 1987.

Atkinson, M.P., Buneman, O.P., and Morrison, R. Binding and type checking in database programming languages. *Computer Journal,* 31(2):99-109, 1988*a*.

Atkinson, M.P., Buneman, O.P.. and Morrison, R., eds. *Data Types and Persistence.* New York: Springer-Verlag, 1988*b*.

Atkinson, M.P., Chisholm, K.J., and Cockshott, W.P. PS-algol: An algol with a persistent heap. *ACM SIGPLAN Notices,* 17(7):24-31, 1982.

Atkinson, M.P., Chisholm, K.J., and Cockshott, W.P. CMS—A chunk management system. *Software: Practice and Experience,* 13(3):259-272, 1983*b*.

Atkinson, M.P., Lécluse, C., Philbrow, P., and Richard, P. Design issues in a map language. In: Kanellakis, P. and Schmidt, J.W., eds. *Bulk Types and Persistent Data,* San Fransisco, CA: Morgan Kaufmann, 1991, pp. 20-32.

Atkinson, M.P., Maier, D., and Benzaken, V., eds. *Persistent Object Systems, Tarascon 1994.* Berlin: Springer-Verlag, 1995.

Atkinson, M.P. and Morrison, R. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems,* 7(4):539-559, 1985.

Atkinson, M.P. and Morrison, R. Integrated persistent programming systems. *Proceedings of the Nineteenth International Conference on Systems Sciences,* Hawaii, 1986.

Atkinson, M.P. and Morrison, R. Polymorphic names, types, constancy and magic in a type secure persistent object store. *Proceedings of the Second International Workshop on Persistent Object Systems,* Appin, Scotland, 1987.

Atkinson, M.P. and Morrison, R. Coordinators of persistent systems track. *Proceedings of the Twenty-second International Conference on Systems Sciences,* Hawaii, 1989.

Atkinson, M.P., Morrison, R., and Pratten, G.D. Designing a persistent information space architecture. *Proceedings of the Tenth IFIP World Congress,* Dublin, Ireland, 1986.

Atkinson, M.P., Sjøberg, D.I.K., and Morrison, R. Managing change in persistent object systems. *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software,* Kanazawa, Japan, 1993*b*.

Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lécluse, C., Pfeffer, P., Richard, P., and Velez, F. The design and implementation of O_2, an object-oriented database system. In: Dittrich, K.R., ed. *Lecture Notes in Computer Science 334,* Berlin: Springer-Verlag, 1988, pp 1-22.

Bancilhon, F. and Buneman, O.P., eds. *Advances in Database Programming Languages.* Reading, MA: Addison-Wesley and ACM Press, 1990.

Bancilhon, F., Delobel, C., and Kanellakis, P., eds. *The Story of O_2: Building an Object-Oriented Database System.* San Francisco, CA: Morgan Kaufmann, 1992.

Beeri, C., Ohori, A., and Shasha, D.E. Database programming languages. *Proceedings of the Fourth International Workshop on Database Programming Languages–Object Models and Languages,* New York, 1993.

Benzaken, V., Delobel, C., and Harrus, G. Clustering strategies in the O_2 object-oriented database system. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/91/21, 1991.

Berman, S. P-Pascal: A data-oriented persistent programming language. PhD Thesis, University of Cape Town, 1991.

Bocca, J. and Bailey, P.J. Logic languages and relational DBMSs—the point of convergence. *Proceedings of the Second International Workshop on Persistent Object Systems,* Appin, Scotland pp 346-362.

Bott, M.F., ed. *ECLIPSE: An Integrated Project Support Environment.* Peter Peregrinus, 1989.

Breazu-Tannen, V., Buneman, O.P., and Naqvi, S. Structural recursion as a query language. *Proceedings of the Third International Workshop on Database Programming Languages,* Nafplion, Greece, 1991.

Bretl, B., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E.H., and Williams, M. The GemStone data management system. In: Kim, W. and Lochovsky, F., eds., *Object-Oriented Concepts, Databases, and Applications,* Reading, MA: Addison Wesley and ACM Press, 1989, pp 283-308.

Brookes, S.D., Hoare, C., and Roscoe, A. A theory of communicating sequential processes. Carnegie-Mellon University Technical Report CMU-CS-83-153, 1980.

Brown, A.L. A distributed stable store. Universities of Glasgow and St Andrews Technical Report PPRR-50-87, 1987.

Brown, A.L. Persistent Object Stores. Ph.D. Thesis, University of St Andrews, 1989.

Brown, A.L. and Cockshott, W.P. The CPOMS Persistent Object Management System. Universities of Glasgow and St Andrews Technical Report PPRR-13-85, 1985.

Brown, A.L., Mainetto, G., Matthes, F., Müller, R., and McNally, D.J. An open system architecture for a persistent object store. *Proceedings of the Twenty-fifth International Conference on Systems Sciences,* Hawaii, 1992.

Brown, A.L. and Morrison, R. A generic persistent object store. *Software Engineering Journal,* 7(2):161-168, 1992.

Brown, J.C. A library explorer for the Napier88 Glasgow libraries. MSc Thesis, University of Glasgow, 1993.

Bruynooghe, R.F., Parker, J.M., and Rowles, J.S. PSS: A system for process enactment. *Proceedings of the First International Conference on the Software Process: Manufacturing Complex Systems,* 1991.

Buneman, O.P., Hirschberg, J., and Root, D. Integrating CODASYL with high level programming languages. *Proceedings of the Second British National Conference on Databases,* Bristol, England, 1982.

Buneman, O.P., Libkin, L., Suciu, D., Tannen, V., and Wong, L. Comprehension syntax. *ACM SIGMOD Record,* 23(1):87-96, 1994.

Burstall, R.M., Collins, J.S., and Popplestone, R.J. Programming in POP-2. Edinburgh University Press, Edinburgh, Scotland, 1971.

Burstall, R.M. and Lampson, B. A kernel language for abstract data types and modules. *Proceedings of the International Symposium on the Semantics of Data Types,* Sophia-Antipolis, France, 1984.

Bushell, S.J., Dearle, A., Brown, A.L., and Vaughan, F.A. Using C as a compiler target language for native code generation in persistent systems. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Campbell, R.H. and Haberman, A.N. The specification of process synchronisation by path expressions. *Lecture Notes in Computer Science 16,* Berlin: Springer-Verlag, 1974.

Cardelli, L. Amber. *Lecture Notes in Computer Science 242,* Berlin: Springer-Verlag, 1986, pp 21-47.

Cardelli, L. Typeful programming. DEC Technical Report 45, 1989.

Cardelli, L. Obliq: A language with distributed scope. *Computing Systems,* 8(1):27-59, 1995.

Cardelli, L. and MacQueen, D.B. Persistence and type abstraction. In: Atkinson, M.P., Buneman, O.P., and Morrison, R., eds. *Data Types and Persistence,* Berlin: Springer-Verlag, 1988, pp 31-42.

Cardelli, L. and Wegner, P. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys,* 17(4):471-523, 1985.

Chan, A., Dayal, U., and Fox, S. An Ada-compatible distributed database management system. *Proceedings of the IEEE, Special Issue on Distributed Databases,* 1987, pp 674-694.

Chiu, S.-Y. and Levin, R. The Vesta repository: A file system extension for software development. DEC Systems Research Center Technical Report 106, 1993.

Clamen, S.M. Schema evolution and integration. *Distributed and Parallel Databases,* 2(1):101-126, 1994.

Cluet, S. and Delobel, C. Towards a unification of rewrite based optimisation techniques for object-oriented queries. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/91/19, 1991.

Cluet, S. and Moerkotte, G. Nested queries in object bases. ESPRIT BRA Project 6309 $FIDE_2$ Technical Report FIDE/93/69, 1993.

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J., and Morrison, R. POMS: A persistent object management system. *Software: Practice and Experience,* 14(1):49-71, 1984.

Connor, R.C.H. Types and polymorphism in persistent programming systems. Ph.D. Thesis, University of St Andrews, 1990.

Connor, R.C.H., Atkinson, M.P., Berman, S., Cutts, Q.I., Kirby, G.N.C., and Morrison, R. The joy of sets. *Proceedings of the Fourth International Conference on Database Programming Languages,* New York, 1993.

Connor, R.C.H., Brown, A.B., Cutts, Q.I., Dearle, A., Morrison, R., and Rosenberg, J. Type equivalence checking in persistent object systems. *Proceedings of the Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, 1990*a*.

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Moore, V.S., and Morrison, R. Unifying interaction with persistent data and program. *Proceedings of the Second International Workshop on User Interfaces to Databases,* Ambleside, Cumbria, 1994*a*.

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., and Morrison, R. Using persistence technology to control schema evolution. *Proceedings of the Ninth ACM Symposium on Applied Computing,* Phoenix, AZ, 1994*b*.

Connor, R.C.H., Dearle, A., Morrison, R., and Brown, A.L. Existentially quantified types as a database viewing mechanism. *Proceedings of the Second International Conference on Extending Database Technology,* Venice, Italy, 1990*b*.

Connor, R.C.H., McNally, D.J., and Morrison, R. Subtyping and assignment in database programming languages. *Proceedings of the Third International Workshop on Database Programming Languages,* Nafplion, Greece, 1991.

Connor, R.C.H. and Morrison, R. Subtyping without tears. *Proceedings of the Fifteenth Australian Computer Science Conference,* Hobart, Tasmania, 1992.

Cook, J.E., Wolf, A.L. and Zorn, B.G. The performance of partitioned garbage collection in object databases. University of Colorado Technical Report CU-CS-653-93, 1993.

Cooper, R.L. Configurable data modeling systems. *Proceedings of the Ninth International Conference on the Entity Relationship Approach,* Lausanne, Switzerland, 1990*a*.

Cooper, R.L. On the utilisation of persistent programming environments. Ph.D. Thesis, University of Glasgow, 1990*b*.

Cooper, R.L., Atkinson, M.P., Dearle, A., and Abderrahmane, D. Constructing database systems in a persistent environment. *Proceedings of the Thirteenth International Conference on Very Large Data Bases,* Location?, 1987.

Cooper, R. and Kirby, G.N.C. Type-safe linguistic run-time reflection: A practical perspective. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Cooper, R.L. and Qin, Z. A graphical data modelling program with constraint specification and management. *Proceedings of the Tenth British National Conference on Databases,* Aberdeen, 1992.

Cooper, R.L. and Qin, Z. A generic data model for the support of multiple user interaction facilities. *Proceedings of the International Conference on the Entity Relationship Approach,* 1994.

Currie, I.F. Filestore and modes in Flex. *Proceedings of the First International Workshop on Persistent Object Systems,* Appin, Scotland, 1985.

Curtis, B., Kellner, M.I., and Over, J. Process modeling. *Communications of the ACM,* 35(9):75-90, 1992.

Cutts, Q.I. Delivering the benefits of persistence to system construction and execution. Ph.D. Thesis, University of St Andrews, 1992.

Cutts, Q.I., Connor, R.C.H., Kirby, G.N.C., and Morrison, R. An execution driven approach to code optimisation. *Proceedings of the Seventeenth Australasian Computer Science Conference,* Christchurch, New Zealand, 1994.

Dahl, O. and Nygaard, K. Simula, an algol-based simulation language. *Communications of the ACM,* 9(9):671-678, 1966.

Davie, A.J.T. and McNally, D.J. Statically typed applicative persistent language environment (STAPLE) reference manual. University of St Andrews Technical Report CS/90/14, 1990.

Daynès, L. and Gruber, O. Customizing concurrency controls using graph of locking capabilities. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Dearle, A. On the construction of persistent programming environments. Ph.D. Thesis, University of St Andrews, 1988.

Dearle, A. Environments: A flexible binding mechanism to support system evolution. *Proceedings of the Twenty-second International Conference on Systems Sciences,* Hawaii, 1989.

Dearle, A. and Brown, A.L. Safe browsing in a strongly typed persistent environment. *Computer Journal,* 31(6):540-544, 1988.

Dearle, A., Cutts, Q.I., and Kirby, G.N.C. Browsing, grazing, and nibbling persistent data structures. *Proceedings of the Third International Workshop on Persistent Object Systems,* Newcastle, Australia, 1990*a*.

Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J., and Vaughan, F. Grasshopper: An orthogonally persistent operating system. *Computer Systems,* 7(3):289-312, 1994.

Dearle, A., Marlin, C.D., and Dart, P. A hyperlinked persistent software development environment. *Proceedings of Hyper-Oz'92: A Workshop on Hypertext Activities in Australia,* Adelaide, Australia, 1992*a*.

Dearle, A., Rosenberg, J., Henskens, F.A., Vaughan, F., and Maciunas, K.J. An examination of operating system support for persistent object systems. *Proceedings of the Twenty-fifth International Conference on System Sciences,* Hawaii, 1992*b*.

Dearle, A., Shaw, G.M., and Zdonik, S.B., eds. *Implementing Persistent Object Bases: Principles and Practice.* San Francisco, CA: Morgan Kaufmann, 1990*b*.

Deux, O. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering.* 2(1), 1990.

Deux, O. The O_2 system. *Communications of the ACM,* 34(10):34-48, 1991.

Dijkstra, E.W. Cooperating sequential processes. In: Genuys, F., ed., *Programming Languages.* Academic Press, 1968*a*, pp. 43-112.

Dijkstra, E.W. The structure of the THE multiprogramming system. *Communications of the ACM,* 11(5):341-346, 1968*b*.

Ellis, C.A. and Gibbs, S.J. Concurrency control in groupware systems. *Proceedings of the ACM-SIGMOD International Conference on the Management of Data,* Portland, OR, 1989.

Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Communications of the ACM,* 19(11):624-633, 1976.

Farkas, A.M. and Dearle, A. Octopus: A reflective language mechanism for object manipulation. *Proceedings of the Fourth International Conference on Database Programming Languages,* New York, 1993.

Farkas, A.M., Dearle, A., Kirby, G.N.C., Cutts, Q.I., Morrison, R., and Connor, R.C.H. Persistent program construction through browsing and user gesture with some typing. *Proceedings of the Fifth International Workshop on Persistent Object System,* San Miniato, Italy, 1992.

Fegaras, L., Sheard, T., and Stemple, D. Uniform traversal combinators: Definiton, use, and properties. *Proceedings of the Eleventh Conference on Automated Deduction (CADE-11),* Saratoga Springs, NY, 1992.

Fegaras, L. and Stemple, D. Using type transformation in database system implementation. *Proceedings of the Third International Workshop on Database Programming Languages,* Nafplion, Greece, 1991.

Fenichel, R.R. and Yochelson, J.C. A LISP garbage collector for virtual-memory systems. *Communications of the ACM,* 12(11):611-612, 1969.

Ghelli, G., Orsini, R., Paz, A.P., and Trinder, P. Design of an integrated query and manipulation notation for database languages. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/41, 1992.

Goldberg, A. and Robson, D. *Smalltalk-80: The Language and Its Implementation.* Reading, MA: Addison Wesley, 1983.

Greenwood, R.M., Guy, M.R., and Robinson, D.J.K. The use of a persistent language in the implementation of a process support system. *ICL Technical Journal,* 8(1):108-130, 1992.

Griswold, R.E., Poage, J.F., and Polonsky, I.P. *The SNOBOL4 Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1971.

Grossman, R.L., Qin, X., Xu, W., Ramamoorthy, H., and Aravjo, N. Managing physical folios of objects between nodes. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Gruber, O. Eos, and environment for persistent and distributed applications over a shared object space. Ph.D. Thesis, Université Pierre et Marie Curie, Paris VI, 1992.

Gruber, O., Amsaleg, L., Daynès, L., and Valduriez, P. Eos, an environment for object-based systems. *Proceedings of the Twenty-fifth International Conference on Systems Sciences,* Hawaii, 1992.

Hammer, M. and McLeod, D. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems,* 6(3):351-386, 1981.

Han, J. and Welsh, J. Methodology modelling: Combining software processes with software products. University of Queensland Software Validation Reasearch Centre Technical Report 93-17, 1993.

Hoare, C.A.R. Monitors: An operating system structuring concept. *Communications of the ACM,* 17(10):549-557, 1974.

Hughes, J.G. and Connolly, M. Data abstraction and transaction processing in the database programming language RAPP. In: Bancilhon, F. and Buneman, O.P., eds. *Advances in Database Programming Languages.* Reading, MA: Addison Wesley and ACM Press, 1990, pp. 177-186.

Hull, R., Morrison, R., and Stemple, D. Database programming languages. *Proceedings of the Second International Workshop on Database Programming Languages,* Salishan Lodge, OR, 1989.

Hurst, A.J. and Sajeev, A.S.M. A capability based language for persistent programming. In: Rosenberg, J. and Koch, D.M., eds. *Persistent Object Systems.* Berlin: Springer-Verlag, 1990, pp. 186-201.

Kanellakis, P. and Schmidt, J.W., eds. *Database Programming Languages: Bulk Types and Persistent Data.* Menlo Park, CA: Morgan Kaufmann, 1991.

Kemper, A. and Kossmann, D. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *VLDB Journal,* 4(3):519-566, 1995.

King, F. IBM report on the contents of a sample of programs surveyed. San Jose, CA: IBM, 1978.

Kirby, G.N.C. Persistent programming with strongly typed linguistic reflection. *Proceedings of the Twenty-Fifth International Conference on Systems Sciences,* Hawaii, 1992*a*.

Kirby, G.N.C. Reflection and hyper-programming in persistent programming systems. Ph.D. Thesis, University of St Andrews, Scotland, 1992*b*.

Kirby, G.N.C., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Moore, V.S., Morrison, R., and Munro, D.S. The Napier88 Standard Library Reference Manual, Version 2.2, University of St. Andrews Technical Report CS/94/7, 1994*a*.

Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M., and Morrison, R. Persistent hyper-progams. *Proceedings of the Fifth International Workshop on Persistent Object Systems,* San Miniato, Italy, 1992.

Kirby, G.N.C., Connor, R.C.H., and Morrison, R. START: A linguistic reflection tool using hyper-program technology. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994*b*.

Kirby, G.N.C. and Dearle, A. An adaptive graphical browser for Napier88. University of St Andrews Technical Report CS/90/16, 1990.

Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin C., Fazakerley, R., and Barter, C. Cache coherence and storage management in a persistent object system. In: Dearle, A., Shaw, G., and Zdonik, S.B., eds. *Implementing Persistent Object Bases.* Menlo Park, CA: Morgan Kaufmann, 1990, pp. 103-113.

Koch, J., Mall, M., Putfarken, P., Reimer, M., Schmidt, J.W., and Zehnder, C.A. Modula/R Report Lilith Version. ETH Zürich, 1983.

Kolodner, E. Recovery using virtual memory. M.Sc. Thesis, University of MIT, 1987.

Kolodner, E., Liskov, B., and Weihl, W. Atomic garbage collection: Managing a stable heap. *Proceedings of the ACM SIGMOD International Conference on the Management of Data,* Portland, OR, 1989.

Krablin, G.L. Building flexible multilevel transactions in a distributed persistent environment. *Proceedings of the Second International Workshop on Persistent Object Systems,* Appin, Scotland, 1987*a*.

Krablin, G.L. Transactions and concurrency. Universities of Glasgow and St Andrews Technical Report PPRR-46-87, 1987*b*.

Kulkarni, K.G. ADT-based type system for SQL. In: Freytag, J.C., Maier, D., and Vossen, G., eds. *Query Processing for Advanced Database Systems.* San Francisco, CA: Morgan Kaufmann, 1994.

Kung, H.T. and Robinson, J.T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems,* 6(2):213-226, 1982.

Lavery, D.O. The design of effective software visualizations for persistent programming languages. ESPRIT BRA Project 6309 FIDE$_2$ Technical Report FIDE/95/116, 1995*a*.

Lavery, D.O. Towards visualizing persistent stores. ESPRIT BRA Project 6309 FIDE$_2$ Technical Report FIDE/95/122, 1995*b*.

Levin, R. and McJones, P.R. The Vesta approach to precise configuration of large softward systems. DEC Systems Research Center Technical Report 105, 1993.

Liskov, B. Distributed programming in Argus. *Communications of the ACM,* 31(3): 300-312, 1988.

Liskov, B., Johnson, P., Gruber, R., and Shrira, L. A highly available object repository for use in a heterogeneous distributed system. *Proceedings of the Fourth International Workshop on Persistant Object Systems,* Martha's Vineyard, 1990.

Matthes, F. Generic database programming: A linguistic and architectural framework. Ph.D. Thesis, University of Hamburg, 1992.

Matthes, F., Müller, R., and Schmidt, J.W. Object stores as servers in persistent programming environments: The P-Quest experience. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/48, 1992.

Matthes, F. and Müβig, S. The Tycoon language TL: An introduction. University of Hamburg Technical Report DBIS 112-93, 1993.

Matthes, F. and Müβig, S., and Schmidt, J.W. Persistent polymorphic programming in Tycoon: An introduction. ESPRIT BRA Project 6309 FIDE$_2$ Technical Report FIDE/94/106, 1994.

Matthes, F. and Schmidt, J.W. The type system of DBPL. *Proceedings of the Second International Workshop on Database Programming Languages,* Salishan, OR, 1989.

Matthes, F. and Schmidt, J.W. Bulk types: Built-in or add-on? *Third International Workshop on Database Programming Languages,* Nafplion, Greece, 1991.

Matthes, F. and Schmidt, J.W. Definition of the Tycoon language TL: A preliminary report. University of Hamburg Technical Report FBI-HH-B-160/92, 1992.

Matthes, F. and Schmidt, J.W. Persistent threads *Proceedings of the Twentieth International Conference on Very Large Data Bases,* Santiago, Chile, 1994.

Matthews, D.C.J. Poly report. University of Cambridge Technical Report 28, 1982.

Matthews, D.C.J. A persistent storage system for Poly. Cambridge University, 1985.

Matthews, D.C.J. Papers on Poly/ML. Cambridge University, 1989.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. *The Lisp Programmers' Manual.* Cambridge, MA: MIT Press, 1962.

Microsoft Corporation. Microsoft Access: Building Applications, 1994*a*.

Microsoft Corporation. Microsoft Access: User's Guide, 1994*b*.

Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences,* 17(3):348-375, 1978.

Milner, R. *Lecture Notes in Computer Science, Vol. 92: A Calculus of Communicating Systems.* Berlin: Springer-Verlag, 1980.

Milner, R. The polyadic π-calculus: A tutorial. University of Edinburgh Technical Report ECS-LFCS-92-249, 1991.

Mira da Silva, M. Automating type-safe RPC. *Proceedings of the Fifth International Workshop on Research Issues on Data Engineering: Distributed Object Management,* Taipei, Taiwan, 1995.

Mira da Silva, M., Atkinson, M.P., and Black, A. Semantics for parameter passing in a type-complete persistent RPC. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/95/124, 1995.

Morrison, R. On the development of Algol. Ph.D. Thesis, University of St Andrews, 1979.

Morrison, R. and Atkinson, M.P. Persistent object systems. *Proceedings of the Twenty-Fifth International Conference on Systems Sciences,* Hawaii, 1992.

Morrison, R., Atkinson, M.P., and Dearle, A. Flexible incremental bindings in a persistent object store. Universities of Glasgow and St Andrews Technical Report PPRR-38-87, 1987*a*.

Morrison, R., Baker, C., Connor, R.C.H., Cutts, Q.I., and Kirby, G.N.C. Approaching integration in software environments. University of St Andrews Technical Report CS/93/10, 1993*a*.

Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A., Hurst, A.J., and Livesey, M.J. Language design issues in supporting process-oriented computation in persistent environments. *Proceedings of the Twenty-second International Conference on Systems Sciences,* Hawaii, 1989*a*.

Morrison, R., Barter, C.J., Connor, R.C.H., Denton, J., Kirkpatrick, G., Munro, D.S., Pretsell, B., and Stemple, D. Concurrency control in process models. *IOPENER,* 2(1):11-12, 1993*b*.

Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., and Dearle, A. On the integration of object-oriented and process-oriented computation in persistent environments. In: Dittrich, K.R., ed. *Lecture Notes In Computer Science 334.* Berlin: Springer-Verlag, 1988, pp. 334-339.

Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A., and Atkinson, M.P. Polymorphism, persistence, and software reuse in a strongly typed object-oriented environment. *Software Engineering Journal,* December, 1987*b*, pp. 199-204.

Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A., and Atkinson, M.P. The Napier type system. *Proceedings of the Third International Workshop on Persistent Object Systems,* Newcastle, Australia, 1990*a*.

Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C., and Munro, D.S. The Napier88 Reference Manual (Release 2.0). University of St Andrews Technical Report CS/94/8, 1994*a*.

Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Dearle, A., Rosenberg, J., and Stemple, D. Protection in persistent object systems. *Proceedings of the International Workshop on Security and Persistence,* Bremen, 1990*b*.

Morrison, R., Brown, A.L., Connor, R.C.H., and Dearle, A. The Napier88 Reference Manual. Universities of Glasgow and St Andrews Technical Report PPRR-77-89, 1989*b*.

Morrison, R., Brown, A.L., Dearle, A., and Atkinson, M.P. An integrated graphics programming environment. *Computer Graphics Forum,* 5(2):147-157, 1986.

Morrison, R., Brown, A.L., Dearle, A., and Atkinson, M.P. On the classification of binding mechanisms. *Information Processing Letters,* 34:51-55, 1990*c*.

Morrison, R., Connor, R.C.H., Cutts, Q.I., Dunstan, V.S., and Kirby, G.N.C. Exploiting persistent linkage in software engineering environments. *Computer Journal,* 38(1):1-16, 1995.

Morrison, R., Connor, R.C.H., Cutts, Q.I., and Kirby, G.N.C. Persistent possibilities for software environments. *Proceedings of the ICSE-16 Workshop on the Intersection Between Databases and Software Engineering,* Sorrento, Italy, 1994*b*.

Morrison, R., Dearle, A., Bailey, P.J., Brown, A.L., and Atkinson, M.P. The persistent store as an enabling technology for integrated project support environments. *Proceedings of the Eighth IEEE International Conference on Software Engineering,* London, 1985.

Morrison, R., Dearle, A., Connor, R.C.H., and Brown, A.L. An ad-hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems,* 13(3):342-371, 1991.

Moss, E. and Sinofsky, A. Managing persistent data with Mneme: Designing a reliable, shared object interface. In: *Advances in Object-Oriented Database Systems.* Berlin: Springer-Verlag, 1988, pp. 298-316.

Moss, J.E.B. Working with persistent objects: To swizzle or not to swizzle. COINS, University of Massachusetts Technical Report 90-38, 1990.

Moss, J.E.B. and Hosking, A.L. Expressing object residency optimizations using pointer type annotations. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Munro, D.S. On the integration of concurrency, distribution, and persistence. Ph.D. Thesis, University of St Andrews, 1993.

Munro, D.S., Connor, R.C.H., Morrison, R., Scheuerl, S., and Stemple, D. Concurrent shadow paging in the Flask architecture. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Mylopoulos, J., Bernstein, P.A., and Wong, H.K.T. A language facility for designing database-intensive applications. *Association for Computing Machinery Transactions on Database Systems,* 5(2):185-207, 1980.

Nettles, S. and Wing, J. Persistence + undoability = transactions. *Proceedings of the Twenty-fifth International Conference on Systems Sciences,* Hawaii, 1992.

Nodine, M.H., and Zdonik, S.B. Cooperative transaction hierarchies: Transaction support for design applications. *VLDB Journal,* 1(1):41-80, 1992.

Ohori, A., Buneman, P., and Breazu-Tannen, V. Database programming in Machiavelli: A polymorphic language with static type inference. *Proceedings of SIGMOD'89,* 1989.

Organick, E.I. *The Multics System: An examination of its structure.* Cambridge, MA: MIT Press, 1972.

Powell, M.S. Adding programming facilities to an abstract data store. *Proceedings of the First International Workshop on Persistent Object Systems,* Appin, Scotland, 1985.

PS-algol. PS-algol reference manual, 4th edition. Universities of Glasgow and St Andrews Technical Report PPRR-12-88, 1988.

Rees, J. and Clinger, W. Revised report on the algorithmic language scheme. *ACM SIGPLAN Notices,* 21(12):37-43, 1986.

Reinwald, B., Dessloch, S., Carey, M., Lehman, T., Pirahesh, H., and Srinivasan, V. Making real data persistent: Initial experiences with SMRC. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Richards, M. and Whitby-Strevans, C. *BCPL: The Language and Its Compiler.* Cambridge, UK: Cambridge University Press, 1979.

Richardson, J. and Carey, M. Persistence in the E language: Issues and implementation. *SPE,* 19(12):1115-1150, 1989.

Richardson, J. and Carey, M. Implementing persistence in E. In: Rosenberg, J. and Koch, D.M., eds. *Persistent Object Systems.* Berlin: Springer-Verlag, 1990, pp. 175-199.

Rosenberg, J. The MONADS architecture: A layered view. *Proceedings of the Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, 1990.

Rosenberg, J. and Dearle, A. Proceedings of Minitrack on distribution and concurrency in persistent systems. *Proceedings of the Twenty-Eighth Conference on Systems Sciences,* Hawaii, 1995.

Rosenberg, J., Henskens, F., Brown, A.L., Morrison, R., and Munro, D. Stability in a persistent store based on a large virtual memory. *Proceedings of the International Workshop on Security and Persistence,* Bremen, Germany, 1990.

Rosenberg, J., and Keedy, J.L., eds. *Security and Persistence.* Berlin: Springer-Verlag, 1990.

Rosenberg, J. and Koch, D. Persistent object stores. *Proceedings of the Third International Workshop on Persistent Object Systems,* Newcastle, Australia, 1989.

Ruffin, M. Kitlog: A generic logging service. *Proceedings of the Eleventh IEEE Symposium on Reliable Distributed Systems,* Houston, TX, 1992.

Russell, G. DOLPHIN: Persistent, object-oriented, and networked. Ph.D. Thesis, University of Strathclyde, 1994.

Russell, G., Shaw, P., and Cockshott, W.P. DAIS: An object-addressed processor cache. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Schaffert, C., Cooper, T., and Wilpot, C. Trellis object-based environment language reference manual. DEC Systems Research Center, 1985.

Scheuerl, S.J.G., Connor, R.C.H., Morrison, R., Moss, J.E.B., and Munro, D.S. MaStA: An I/O cost model for database crash recovery mechanisms. University of St Andrews Technical Report, CS/95/1, 1995.

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

Schmidt, J.W. and Matthes, F. The database programming language DBPL rationale and report. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/46, 1992.

ServioLogic Ltd. Programming in OPAL. 1987.

Shapiro, M., Gautron, P., and Mosseri, L. Persistence and migration for C++ objects. *Proceedings of the European Conference on Object-Oriented Programming,* 1989.

Sheard, T. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems,* 19(4):531-557, 1991.

Sheard, T. and Hook, J. Type safe meta-programming. Oregon Graduate Institute, 1994.

Sheard, T. and Stemple, D. Automatic verification of database transaction safety. *ACM Transactions on Database Systems,* 12(3):322-368, 1989.

Shipman, D. The functional data model and the data language DAPLEX. *Association for Computing Machinery Transactions on Database Systems,* 6(1):140-173, 1981.

Singhal, V., Kakkad, S.V., and Wilson, P.R. Texas: An efficient, portable, persistent store. *Proceedings of the Fifth International Workshop on Persistent Object Systems,* San Miniato, Italy, 1992.

Sjøberg, D.I.K. The Thesaurus: A tool for meta data management. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/91/6, 1991.

Sjøberg, D.I.K. Thesaurus-based methodologies and tools for maintaining persistent application systems. Ph.D. Thesis, University of Glasgow, 1993.

Sjøberg, D.I.K., Atkinson, M.P., Lopes, J.C., and Trinder, P.W. Building an integrated persistent application. *Proceedings of the Fourth International Conference on Database Programming Languages,* New York City, 1993.

Sjøberg, D.I.K., Atkinson, M.P., and Welland, R.C. Thesaurus-based software environments. *Proceedings of the ICSE-16 Workshop on the Intersection Between Databases and Software Engineering,* Sorrento, Italy, 1994*a*.

Sjøberg, D.I.K., Cutts, Q.I., Welland, R., and Atkinson, M.P. Analysing persistent language applications. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994*b*.

Sjøberg, D.I.K., Philbrow, P.C., Waite, C., and Welland, R. Build management in database programming language environments. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/95/123, 1995.

Smith, J.M., Fox, S., and Landers, T. Reference Manual for ADAPLEX. Computer Corporation of America Technical Report CCA-81-02.

Sommerville, I., Welland, R., Potter, S., and Smart, J. The ECLIPSE user interface. *Software: Practice and Experience,* 19(4):371-391, 1989.

Stanley, M. An evaluation of the Flex PSE. Defence Research Agency, Malvern, England Technical Report 86003, 1986.

Stanley, M. and Drummond, P.D. A flexible basis for software configuration management. Defence Research Agency, Malvern, England Technical Report 4127, 1988.

Stemple, D., Fegaras, L., Sheard, T., and Socorro, A. Exceeding the limits of polymorphism in database programming languages. In: Bancilhon, F., Thanos, C., and Tsichritzis, D., eds. *Lecture Notes in Computer Science 416,* Berlin: Springer-Verlag, 1990, pp. 269-285.

Stemple, D. and Morrison, R. Specifying flexible concurrency control schemes: An abstract operational approach. *Proceedings of the Fifteenth Australian Computer Science Conference,* Hobart, Tasmania, 1992.

Stemple, D., Sheard, T., and Fegaras, L. Linguistic reflection: A bridge from programming to database languages. *Proceedings of the Twenty-fifth International Conference on Systems Sciences,* Hawaii, 1992*a*.

Stemple, D., Stanton, R.B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G.N.C., Fegaras, L., Cooper, R.L., Connor, R.C.H., Atkinson, M.P., and Alagic, S. Type-safe linguistic reflection: A generator technology. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/49, 1992*b*.

Stonebraker, M. and Kemnitz, G. The POSTGRES next generation database management system. *Communications of the ACM,* 34(10):78-92, 1991.

Strachey, C. *Fundamental Concepts in Programming Languages.* Oxford University Press, Oxford, UK, 1967.

Straw, A.F., Mellender, F., and Riegel, S. Object management in a persistent Smalltalk system. *SPE,* 19(8):719-737, 1984.

Sutton, S. A flexible consistency model for persistent data in software-process programming. *Proceedings of the Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, MA, 1994.

Suzuki, S., Kitsuregawa, M., and Takagi, M. An efficient pointer swizzling method for navigation intensive applications. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994.

Teitelbaum, T. and Reps, T. The Cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM,* 24(9):563-573, 1981.

Tennent, R.D. Language design methods based on semantic principles. *Acta Informatica,* 8:97-112, 1977.

Thatte, S.M. Persistent memory: A storage architecture for object oriented database systems. *Proceedings of the ACM/IEEE International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, 1986.

Thomas, I. PCTE Interfaces: Supporting Tools in Software-Engineering Environments. *IEEE Software,* pp. 15-23, 1989.

Trinder, P.W. and Wadler, P.L. Improving list comprehension database queries. *Proceedings of TENCON'89,* Bombay, India, 1989.

van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A. Report on the algorithmic language ALGOL 68. *Numerische Mathematik,* 14:79-218, 1969.

Wai, F. Distributed concurrent persistent languages: An experimental design and implementation. Universities of Glasgow and St Andrews Technical Report PPRR-76-89, 1989.

Waite, C. et al. The Glasgow persistent workshop: User documentation. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/95/125, 1995.

Wasserman, A.I., Sherertz, D.D., Kersten, M.L., van de Reit, R.P., and Dippé, M.D. Revised report on the programming language PLAIN. *ACM SIGPLAN Notices,* 5(16):59-80, 1981.

Wetzel, I. Programming with STYLE: On the systematic development of programming environments. Ph.D. Thesis, University of Hamburg, Germany, 1994.

Wileden, J., Wolf, A., Fisher, C., and Tarr, P. PGRAPHITE: An experiment in persistent typed object management. *Proceedings of the ACM SIGSOFT '88: Third Symposium on Software Development Environments,* pp 130-142, 1988.

Wilson, P.R. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. University of Illinois at Chicago Technical Report UIC-EECS-90-6.

Wilson, P.R. Uniprocessor garbage collection techniques. *Proceedings of the International Workshop on Memory Management,* St. Malo, France, 1992.

Wirth, N. The programming language Pascal. *Acta Informatica,* 1:35-63, 1971.

Zezula, P. and Rabitti, F. Navigation index for an object store. *Proceedings of the Twenty-fifth International Conference on System Sciences,* Hawaii, 1992.