Managing Change in Persistent Object Systems[*]

Atkinson, M.P.[†] Sjøberg, D.I.K.[‡] Morrison, R.[§]

Abstract

Persistent object systems are highly-valued technology because they offer an effective foundation for building very long-lived *persistent application systems* (PAS). The technology becomes more effective as it offers a more consistently integrated computational context.

For it to be feasible to design and construct a PAS it must be possible to incrementally add program and data to the existing collection. For a PAS to endure it must offer flexibility: a capacity to evolve and change. This paper examines the capacity of persistent object systems to accommodate incremental construction and change.

Established store based technologies can support incremental construction but methodologies are needed to deploy them effectively. Evolving data description is one motivation for inheritance but inheritance alone is not enough to support change management.

The case for supporting incremental change is very persuasive. The challenge is to provide technologies that will facilitate it and methodologies that will organise it.

This paper identifies change absorbers as a means of describing how changes should propagate. It is argued that if we systematically develop an adequate repertoire of change absorbers then they will facilitate much better quality change management.

1 Introduction

The primary interest in object technology arises from its capacity to be an essential material from which large and long-lived application systems are built. Such long-lived applications are called *Persistent Application Systems* (PAS). Examples of such systems are CAD systems, geographic information systems, urban planning systems, health-care management systems, etc. They are characterised by becoming large, often being distributed with a wide variety of users and being concerned with the *long-term* support of cooperative activity. Large investments are involved in their construction and operation. People and organisations depend on them.

[*]Invited paper, JSSST International Symposium on Object Technologies for Advanced Software (Kanazawa, Japan, 4th–6th November 1993)

[†]Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, Scotland

[‡]Institute of Informatics, P.O. Box 1080 Blindern, N-0316 Oslo, Norway

[§]Department of Computational Science, University of St Andrews, St Andrews, KY16 9SX, Scotland

As PAS attain central importance to an organisation (for example, a hospital trusts all its medical histories, accounting information and staff records to a health-care management system) the continuance of the organisation and the adaptability and continuity of the PAS become strongly interrelated. If the PAS fails the organisation may also fail. Equally, if the PAS cannot be adapted to the changing environment and needs of the organisation sufficiently quickly and economically it will inhibit the organisation from adapting. Such rigidity may lead to the demise of the organisation. This paper assumes that PAS will be built using object technologies and enquires about the adequacy of their provisions for change.

Persistent object storage enables a wide range of program and data forms to be built and to endure for as long as the storage technology which holds them and the execution technology which interprets them continue to exist. The stability of references based on identity allows long-term representation of constructional relationships. The binding mechanisms allow incremental growth of the total body of program and other data. The mechanisms of inclusion polymorphism (loosely, inheritance and subtyping) allow certain kinds of change to be accommodated and localised. Hence, at first sight, we might be complacent and regard existing persistent object systems as adequate. This paper attempts to assess precisely where that complacency is well founded and where advances are necessary before organisations may safely become dependent on persistent object technology, or more precisely on PAS built using persistent object technology.

The paper is divided into two parts. In the first part (Sections 2 and 3) the progress towards achieving a consistent platform on which to build PAS is examined. This is very much a sample of actual progress, focusing on orthogonal persistence and the relationship between data models and type systems. Two recent developments in type systems are examined: the provision of parametric type constructors (3.3.1) allows modelling and descriptive capacity to be extended, and the provision of infinite unions (3.3.2) permits certain forms of change absorber to be defined.

The second part of the paper deals with various aspects of change management. Section 4 restates why change management is essential and proposes that improved methods will emphasise incremental change. Section 5 identifies what might be expected from change management tools, how they may be improved with persistent object technology and why they will remain deficient until better structural description is available. Section 6 presents change absorbers as a means of providing this description. Three categories are proposed: automatic transformers (6.1), partial transmitters (6.2) and active responders based on enquiry mechanisms (6.3).

Current practice for managing program change is reformulated in terms of change absorbers in Section 7. It is postulated that the necessary primitives, e.g. dynamic binding and incremental linking may already exist, but regular patterns for their use need to be better characterised. Section 8 shows that coping with type/schema change is a good deal more problematic. The infinite union types that currently exist and type subsumption provide only a partial solution. This leads to a conclusion that identifies research challenges.

2 Progress towards Uniformity

Persistence is provision for values to remain computationally available for an arbitrary length of time; as long as they are required for computation. This period may be very brief or the full life-time of a PAS. Note that, for the PAS envisaged, these maximum life-times may be tens or even hundreds of years.

Orthogonal persistence is the provision of persistence equitably to all values [Atkinson *et al.*, 1982]. It is important as it ensures that PAS designers and programmers may choose data structures freely from all those available. If persistence isn't orthogonal, they will need to choose representations suitable for processing and other representations for storage. This additional modelling complexity and the inevitable translations between representations that result from a failure to make persistence orthogonal are needless impediments to the construction and maintenance of PAS.

It should be noted that the choice of the type system is a separate issue. It might be based entirely on relations, be appropriate only to formatted text processing or be based on the type system of an object oriented programming language. That choice will be made to provide appropriate modelling capabilities for the application area.

A further principle that guides the provision of persistence is that of *persistence independence*. This means that code should not need to be different for data of different age or longevity. A consequence of this principle is that programmers should not be required to explicitly move data from long-term storage to processing storage. Transfers must be automated.

Avoiding the need for translation and explicit transfer reduces the volume of code in a typical PAS by about one third. A more important gain from orthogonal persistence is the reduction of cognitive load on designers and programmers. In a typical system (shown in Figure 1) they are trying to envisage, implement and manage three mappings. They have the particularly tricky task of keeping any pair of the mappings consistent with the other one.

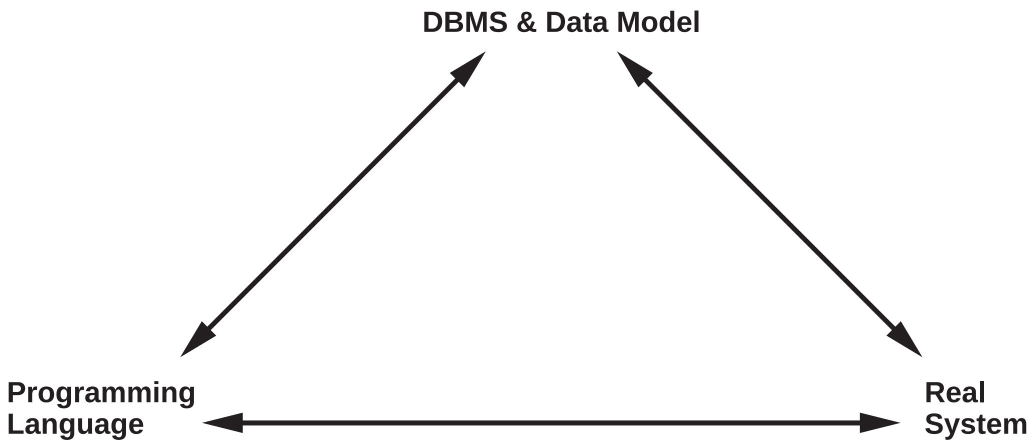

Figure 1: Many PAS are constructed using two models: one in the database and one in the programs

This complexity and the need for programmers to maintain this consistency are both avoided with orthogonal persistence as only one model is used and one mapping maintained (see Figure 2).
Those object-oriented systems that use different notations to describe schemata from those they use to describe data in their operations (methods), fail to deliver this orthogo-

Persistent Programming Language
Real System

Figure 2: New PAS may be constructed using only one model in a PPL

nality. Such object-oriented systems re-introduce the complexity piecemeal as fragmented triple mappings for each operation.

The successful persistent object systems are therefore likely to be those that take some programming language and provide it with orthogonal persistence [Atkinson *et al.*, 1983] or those that are designed explicitly to provide orthogonal persistence [Morrison *et al.*, 1989]. The most popular versions of persistent systems at present are derived from C^{++} [Richardson and Carey, 1987; Bancilhon *et al.*, 1992; Object Design Inc., 1991; Ontologic Inc., 1991].

Provision of orthogonal persistence may be viewed as a step in the process of providing a more uniform computational environment. If the environment in which PAS computation takes place is made uniform several advantages may be expected:

1. programming will be simpler (and hence more economic and less error prone) as programmers have less detailed rules to comprehend;
2. programs will have a more easily specified environment (and therefore it is more reasonable to assume that it can be perpetuated unchanged);
3. incremental construction is simplified as the successive components all interact with a constant environment; and
4. it may be easier to plan and conduct changes to the support platform when it is conceived as a coherent entity.

The search for a uniform computational environment still has a long way to go. For example, Figure 3 shows the complex environment of programs in a recently constructed health management system. Figure 4 shows a much simplified environment based on a hypothetical platform which provides the full range of requirements for PAS components. The search for a uniform scalable platform has been discussed elsewhere [Atkinson, 1992]. For the rest of this paper it is assumed that the optimal context for incremental design and construction will be a uniform and complete platform. Researchers and industry will need to pursue its provision.

3 The Rôle of Types

Throughout the last two decades database researchers have made considerable progress with data models while programming language researchers have made significant progress with type systems. It is helpful to review aspects of that progress to recognise the common goals. This is important if we espouse uniformity of computational environment, as to achieve it requires that *only one* of these is part of that environment.

Type systems will be examined first and then their relationship with data models will be reviewed. A type system services four interrelated tasks:

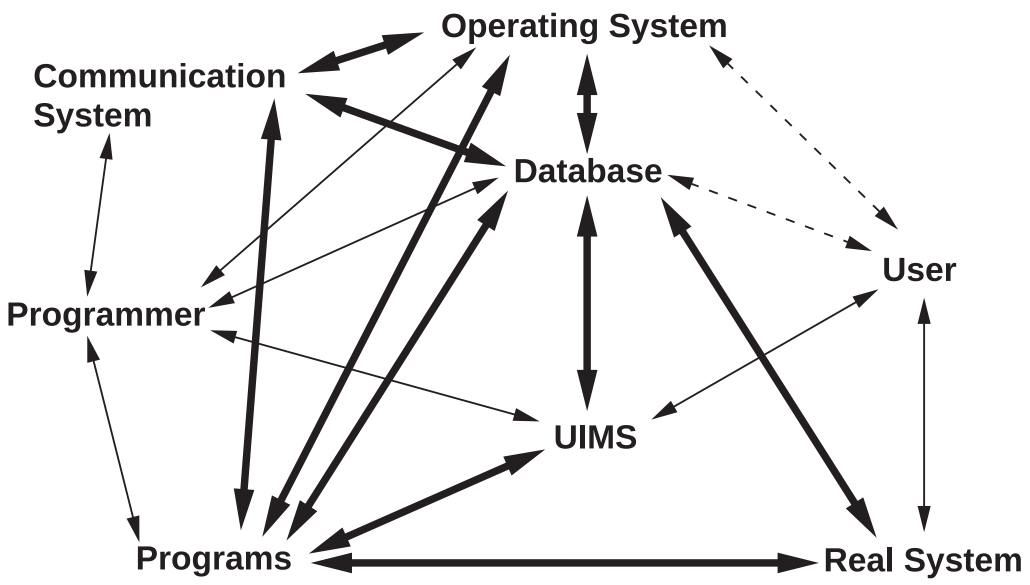

Figure 3: A Recent PAS shows the Complex Environment of its Programs

1. description of the data;
2. restriction of programs to reduce errors;
3. protection of critical structures; and
4. provision of information to implementations.

3.1 Description of Data

A type system provides a means of describing the forms values may take. Typically this may involve a (possibly recursive) combination of the following:

1. choice of some predefined forms, called *base types* (e.g. integer, boolean, real, string, date, etc.);
2. composition of types using *type constructors* to register relationships between values (e.g. aggregations in records, alternatives in unions or tagged sums, collections in sets, mappings in maps and arrays, etc.);
3. *reference*, usually defined via *recursive types*, which implies instances and identity;
4. declaration and naming of new (usually parametric) type constructors;
5. abstract data types, that provide encapsulation; and
6. combination of data structures with program (e.g. operations or procedures) that will manipulate the data in those structures.

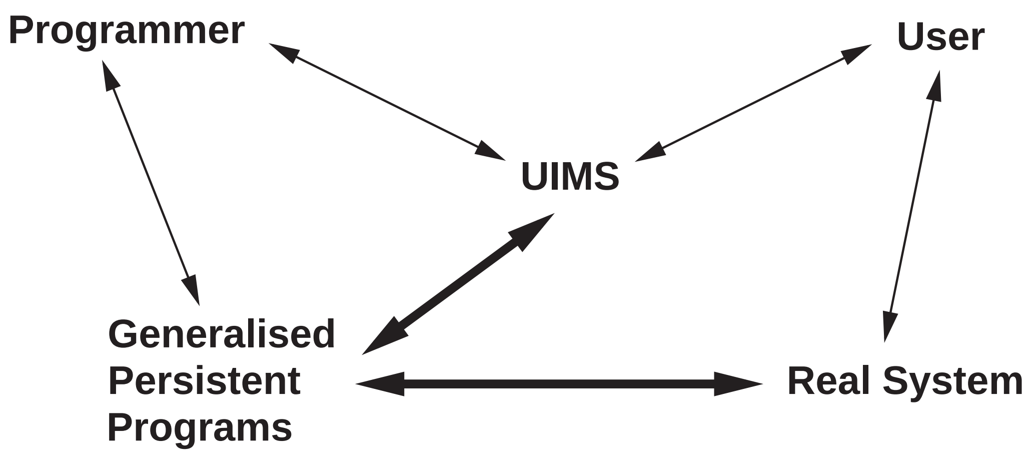

Figure 4: An Ideal Goal for the Computational Environment of PAS Components

Many programming languages provide particular forms and additional forms of the mechanisms listed above; the **env** and **any** types used later (see Sections 3.3.2 and 8) are examples.

Some languages provide a means of defining one type as a refinement of a previously defined type, e.g. Simula [Birtwistle *et al.*, 1973], SmallTalk [Goldberg and Robson, 1983], C^{++} [Ellis and Stroustrup, 1990], Galileo [Albano *et al.*, 1985], Oberon [Reiser, 1991], Fibonacci [Albano *et al.*, 1993], etc. This facility will be called "type inheritance" in this paper, though some call it "explicit subtyping". These languages, and some others e.g. Machiavelli [Ohori *et al.*, 1989], Quest [Cardelli, 1990], Tycoon [Matthes, 1992], etc., permit values of one type to be used where values with less properties are required. This property of types will be called "subtyping" in this paper, though it is more properly called "inclusion polymorphism". Where it isn't combined with type inheritance it may be referred to as "implicit subtyping" [Connor *et al.*, 1990a].

A number of languages (e.g. SETL2 [Schwartz *et al.*, 1986], Pascal/R [Schmidt, 1977], Modula/R [Koch *et al.*, 1983], DBPL [Matthes *et al.*, 1992], P-Pascal [Berman, 1991], RAPP [Hughes and Connolly, 1990], etc.) provide a variety of bulk type constructors, e.g. set, relation, sequence, map, etc. as a means of constructing collections.

This exploration of descriptive components out of which a model can be built is reminiscent of the contemporaneous exploration of data models, e.g. SDM [Hammer and McLeod, 1981], Daplex [Shipman, 1981], RM/T [Codd, 1979], IFO [Abiteboul and Hull, 1987], etc.

The crucial common goal is to impose a structure on the data by mapping it to a composition of standard but flexible, appropriate and suggestive building blocks. The results are useful for several reasons:

- they enable parts or features of the *structure* to be named, thus allowing people and programs to refer to them;
- they give access to operations that are associated with the component structures, for instance, iteration over the elements of some collection and tests on the current form of variants; and

- they facilitate communication among a system's designers, builders and users via related textual and diagrammatic representations.

The programming languages have supported a model of reference and identity and seen it as an important modelling tool for many years [Hoare, 1975; Atkinson, 1978]. They have had a concern for specifying the relationship between code and data, for example, in Fortran subprograms and in modules in many languages. Higher-order languages[1], languages with modules and languages with abstract data types all provide a means of describing a static relationship between program and data. (More flexible relationships will be considered later in the paper — see Section 3.3.2.)

In recent years Object-Oriented Databases (OODBs) have attempted to support similar modelling structures [Atkinson *et al.*, 1989].

3.2 Equivalences between Data Models and Type Systems

The preceding discussion shows that similar activities and common goals can be recognised in data model research and in type system research. Approximate equivalences are summarised in the following table:

Database Vocabulary	Programming Language Vocabulary
data models	type systems
schema	type expression
database	variable
database extent	value

Although detailed comparison shows that in both cases features[2] are tagged onto the concepts that invalidate these equivalences, they hold well at a conceptual level. Therefore it is feasible that one of type systems or data models will fulfill the common rôle. Of importance here is current type system research into methods of describing bulk types [Atkinson *et al.*, 1993]. These allow regular structures to be described and provide operators that abstract over iteration.

3.3 Extensible and Universal Type Constructs

Three areas of innovation now increase the power of type systems:

1. regular and powerful polymorphisms [Cardelli and Wegner, 1985];
2. mechanisms for handling parametric type definitions [Cardelli, 1989]; and
3. models of infinite union types which support incremental evolution and binding [Morrison, 1979].

The latter two of these are now illustrated in a little more detail. It is assumed that the reader is already familiar with parametric and inclusion polymorphism. For a fuller discussion of type system research applicable to persistent application systems the reader is referred to [Connor, 1991; Connor, 1993].

[1]Those that have procedures in their value space, so that procedures may be the arguments to, and results of, other procedures; may be elements of data structures and values of variables.

[2]Typically those concerned with efficiency, such as placement or representation annotations.

3.3.1 Defining New Type Constructors

A type constructor is a programming language or data model construct which when supplied with suitable parameters generates a new type. Readers will be familiar with built-in type constructors, such as **array** and **record** in Pascal-like languages, **table** in SQL and **set** and **sequence** in semantic data models and OODBs. The set of built in constructors is inevitably a compromise between complexity and completeness. A compromise may be unsatisfactory for everyone: the suppliers of the system find it overly complex to build, the users find it overly complex to understand but still lacking in constructs they would like to use in their particular application.

Parametric type definitions allow a programmer to define new constructors in terms of already provided and user-defined constructors. These can be equivalent to those that have already proved their utility in data model research or can be new constructors appropriate to particular application domains.

Where type constructors require parameters these may be supplied with: a base type, another constructed type or with another type constructor. In the last case a new type constructor is defined, in the other cases a type is the result. This will be illustrated by developing an example that describes a database of a teaching department.

type Student **is record**[MatricNo: **integer**; Name: **string**]
type Students **is** Set[Student]
type Course **is record**[Cname: **string**; Max_enrolment: **integer**]
type Courses **is** Set[Course]
type Prerequisites **is** DAG[Course]
type Enrolment **is record**[course: Course; start: Date; enrolled: Students]
type YearsWork **is** Set[Enrolment]
type History **is** Sequence[YearsWork]
type TreadMill **is** Ring[Courses]

Four different constructors have been used: *Set*, with its usual mathematical meaning but probably an implicit order; *DAG*, a directed acyclic graph; *Sequence*, again with its standard meaning and *Ring*, with the obvious meaning. The composition is well illustrated by the type *History* which is itself a *Sequence* of *Sets*, each element of these *Set*s contains a *Set* of students.

Of course, such composition can be a property of any data model that provides *orthogonality* and *data type completeness*; that is, types constructed with any constructor may be parameters to any other constructor. The additional important feature is that these constructors may themselves be defined (out of other constructors). So, while *Set* and *Sequence* might be provided, it is scarcely credible that *DAG* and *Ring* would already be provided. The importance of the current work on types is that they can be defined and then re-used elsewhere. For example, the *DAG* constructor might be used to model descendants in a genetics experiment and *Ring* might be used to model the phases in the life cycle of species in a palaeo-zoological data collection.

The new constructors can be defined by composing those already defined as is illustrated in this example:

type Chain[E] **is** Sequence[Ring [E]]
type Forest[T] **is** Set[DAG[T]]

type Pairs[A, B] **is** Sequence[**record**[first: A; second: B]]

The identifier E denotes a type parameter which is supplied when a type is constructed. Thus an expression like:

type Citations **is** Forest[**string**]

occurring at any subsequent point will be equivalent to:

type Citations **is** Set[DAG [**string**]]

The ability to establish new constructors is used later in the paper.

3.3.2 Infinite Unions and Extensibility

Some type systems provide types that denote an infinite union of types, that is a value that is described by one of these types has an infinite number of possible types from which it was injected into the infinite union. In this context, the type information must be associated with the value to ensure that when it is eventually projected out of the union it is treated in a way compatible with the original type; otherwise, type safety is compromised. A typical name for this type is **any**.

Such types are important as they allow type-safe program to be written that will perform operations on values whose types have not even been defined or thought of at the time of writing. A typical example is a browser [Dearle and Brown, 1988]. Other examples would be form generators or other standard interfaces. The reader should note that this is fundamentally different from the code which can be written using parametric polymorphism alone.

These infinite union types permit a new kind of incremental program development that will be motivated and discussed below. Essentially, a type-check is postponed until projection out of the infinite union occurs. This means that for any code that does not need to unpack the value, type checking is satisfied without enquiry into the type of the injected value. When projection occurs, the system has to examine the type and ensure that the code which will be applied to the projected value only carries out operations compatible with that value. Hence the type checking (correctness validation) has been partitioned into parts:

1. the code prior to the projection which can be checked completely statically;
2. the code after the projection which can be checked completely statically on the assumption that the projection performs satisfactorily; and
3. the projection itself, which is checked dynamically at the time when the projection is performed.

This will be made concrete with another infinite union construct, **env**, in the programming language Napier88 [Dearle, 1989; Morrison *et al.*, 1989] which will be used for illustrative purposes below.

The type **env** denotes any set of bindings. Each binding is a quadruple: a unique identifier, a type, a value, and a constancy. The value must conform to the type. Values of type **env** are therefore a set of such bindings and have all the rights of any value in the language. Operations exist to construct new instances of this type, to add bindings to an **env** value, to remove them, to extract the value, to assign a new value, and to scan all the bindings.

The projection operation takes the form:

```
use e with x:int; p: proc( int ) in
    begin
    p( x )
    ...
    end
```

where the code between the **begin**, **end** pair is statically type checked as is the code outside the **use** statement. However, when the expression e is evaluated (it is statically determined that it must produce an **env**) there is a dynamic check to establish that the particular environment produced actually contains (at least) the bindings specified in the signature after **with** which have been assumed in the compilation of the clause after **in**.

This mechanism is an example of developing support for incremental program development. Such mechanisms allow programs providing definitions, services or values to be developed and replaced independently of the programs which will use those components.

Before discussing other incremental mechanisms, it is important to review (in Section 4) why incremental approaches to large scale system design and maintenance are becoming important.

3.4 Summary of the Provision of Uniform Technology

The case for providing a uniform technology is that it simplifies the processes of PAS design, construction and maintenance. It will also facilitate incremental methods for building and maintaining PAS. The relationship between type systems and data models provides an example of technology which could be made to converge to provide a more uniform computational platform for PAS. For the purposes of this paper, type systems will be used as the putative limit of that convergence. They are chosen as they already have notations for regular extensibility (Section 3.3.1) and incremental binding (Section 3.3.2).

4 Incremental Design and Construction

Incremental construction is one of the traditional reasons for using databases. The database is a central repository and the suites of application programs attached to it are added or replaced incrementally. Schema editors have also provided a mechanism for incrementally extending and changing the definition of the total body of data.

Incremental design is the primary motivation for object orientation. Initial, quite general object specifications (classes) are refined to provide specialisations or to capture increments in the understanding of the modelled system.

These may be seen as examples of a more general principle:

> To be effective all technologies and methodologies for PAS construction must support incremental design, incremental construction and incremental change.

Any PAS will have been developed to support human activity in an organisation. Since human behaviour is sophisticated and in organisations particularly complex, the requirements for a PAS are inevitably sophisticated and complex. It is infeasible to expect that they can be comprehended and converted into a design in a single effort. Early database methodology, in contrast, expected to fully understand an organisation and develop a schema that "represented the enterprise". Experience has shown that for modern requirements it is necessary to understand and design the PAS incrementally. A portion of the enterprise is understood, the corresponding design is developed and the construction is often undertaken before other parts of the enterprise are tackled. This is also a common strategy for organising other large engineering endeavours, e.g. chemical plant design and construction.

It usually transpires that the initial understanding of requirements, even for these parts is inadequate and further revision of the design is undertaken even before the relevant subsystem is complete. Again, it is partly the limitation of implementors' conceptual capacities that engenders this incrementalism. It is almost impossible to fully understand even a part of the activity to be supported and an iterative approach results. System development methodologies normally recognise this, insisting, for example, on dialogue with representatives of all those who will eventually use the PAS, to detect and correct such misunderstandings.

This dialogue and the introduction of parts of the system, is however, a further stimulus to change. It is therefore normal to find that the user community immediately perceives ways in which their operations and the system might now be improved.

However carefully this design process is undertaken, the system continues to require change. Existing operational practices are revised and elaborated and new activities are added. This is a manifestation of a strength of cooperating humans, they continually review and revise their working practices in order to improve their performance or product. This has long been understood.

For example, it was traditional to arrange workers in craft-villages. In Burma, as you travel west from Mandalay to Sagaing, you pass through a village of woodcarvers, a village of people making alabaster casts, a village of bronze casters and a village of gold leaf makers. Clustering the people resulted in more rapid improvement of their technology as they copied each other's improvements and further developed them. A process of incremental change.

As Persistent Application Systems are built it is important that they do not inhibit this process of incremental improvement. Indeed, modern systems make possible the mutual stimulus towards improvement between cooperating workers that are geographically distributed — a "distributed craft-village". To respond, it is necessary that the PAS be able to be incrementally changed both rapidly and economically. There is a danger that massive databases and software will stultify change by embedding the organisation in a mass of "digital concrete". To combat this danger the PAS technologies and methodologies must be developed to support incremental change as a matter of course.

There are other more mundane reasons for espousing incrementalism as a matter of course. Organising design and construction becomes more manageable. The work can be split into separate, relatively small components that match the capacities of small teams. Teams with different expertise, such as requirements, design, implementation and testing, can move ahead of one another preparing the ground for the next team while it is working on another increment.

As PAS and the organisations they support become both more extensive and more intertwined so the feasibility of making radical changes is reduced. It becomes progressively harder to turn off an old system and transfer work to a new one. It becomes impracticable to stop a system for hours or even days while schemata are re-organised and software is replaced.

There is, therefore, an unassailable case for founding the technology and methodology that supports PAS on the observation that change is normal throughout the PAS's lifetime of design, construction, maintenance and operation. Incremental change is easier to manage and probably easier to sustain than cataclysmic change. The challenge is to develop methodologies and technologies that support incremental change well in the context of the very large and very long-lived systems which we aspire to achieve. The following sections assess our current levels of success and identify the challenge more precisely.

5 Change Management Requirements

Change management is concerned with the organisation of change. If the change is to be incremental and if the system is to go on functioning correctly for its users there are two requirements to be met:

1. all the consequences of a change must be properly propagated; and
2. no unnecessary changes should be made.

These requirements will be illustrated with an example from a health care system. It has previously been the practice to record a patient's next-of-kin. It is now noted that for many elderly patients the next-of-kin is unable to perform some required actions as they too are elderly. A new fact about a patient is therefore to be recorded, the responsible-person (perhaps a care-worker, the organiser of a home in which the patient resides, etc.).

The schema is changed to accommodate the extra information. All extant records will not have that extra information and it will not be collected unless or until the patient re-presents. Therefore, it is probably desirable not to change the existing records until the new information becomes available (this also avoids an initial value problem).

Similarly, most programs will not be concerned with handling the new information and should operate unchanged in the future. However, certain programs must be changed, in particular, those that enter a new patient into the system, those concerned with checking the continued correctness of patient details and those that generate letters that should now go to the responsible-person.

These program changes will also require that certain screen designs need to be changed, certainly those used for inputting and checking patient details and at least some of those for displaying patient details in other circumstances. But screen re-design may overload a screen so that the information must be re-organised and split between two screens. This results in a change to the sequence of screen operations. Similarly pro forma, report formats and letter templates may have to be changed.

Some of these changes will affect users. For example, staff admitting patients now have to ask extra questions and enter extra data. The relevant existing staff will need re-training, user documentation will need to be revised, training manuals and documentation of hospital procedures will also need revision. Design and maintenance documentation will also need to be revised.

We may enquire how common such changes are, how much of the system is affected by each change, etc. Recently Sjøberg [Sjøberg, 1993a] has measured such parameters for 18 months in the development of a health care system. There are not many similar studies, so it is impossible to be sure how generally applicable are his observations. However, his study detected very high rates of change and extensive consequential changes. Hopefully, similar studies will soon be available to allow relevant norms to be included in the estimation of development and operational costs.

Sjøberg's data supports anecdotal evidence that it is worthwhile investing in technology that supports the change management process and in methods that make implementing the changes less error prone and costly. There are at present two levels of support:

1. informative systems that provide PAS developers and maintainers with data about the existing system, its present representation and perhaps some of its dependencies; and

2. automatic systems that directly implement some of the steps necessary to deal with the consequences of change.

Examples of the former are data dictionaries, repositories and build management tools. Examples of the latter are automatic forms interfaces that, at the cost of not having a tailored screen design, automatically generate a user interface and automatic re-structuring tools that propagate change to the existing data.

Sjøberg has also implemented a prototype change management system that utilises the reliable references available in persistent systems. As a result of this reliability, his system is able to discover and record dependency information automatically and provide reliable information about the consequences of changes [Sjøberg *et al.*, 1993]. He is also investigating the automation of some aspects of change propagation based on the more reliable information [Sjøberg, 1993b]. However, automation is limited by the lack of information about which changes should be propagated and which absorbed. It is here suggested that this additional information may be provided by a combination of two techniques:

1. the division of the system into cells which communicate through well defined interfaces; and

2. the use of *change absorbers* to specify more precisely the system architect's intent about when and how change should propagate.

A methodology is then needed to ensure that there is disciplined and sufficient provision of the propagation specification information. Given this additional information, it should be possible to use techniques such as those developed by Sjøberg to provide much more helpful information for those considering a change and to automate more of the change processes.

6 Change Absorbers

An important adjunct of change management tools are *change absorbers*. Without them the extent to which change would propagate would be intolerable. Most users want the illusion that the system is unchanging except on the (what they consider rare) occasions when they want a change or they wish to exploit a new feature.

Typical examples of change absorbers are:

1. database views, which protect programs using the view from changes in the schema[3];
2. type subsumption in object oriented systems which allows procedures to continue to operate with refinements of the data types originally expected; and
3. infinite union types which protect part of a type or schema definition from "seeing" changes beyond the infinite union.

A comprehensive repertoire of change absorbers will be needed to permit fluent incremental change. The important property of a change absorber is that it transmits the changes that are important across a boundary and suppresses propagation of all other changes.

The system can then be visualised as a composition of cells. Each cell would be protected by a semi-permeable membrane constructed from change absorbers that transmitted change important to the cell and kept out extraneous change. Such a structure would be recursive, compositions of cells behaving as a cell. The cellular analogy is intended to suggest a total system in which its smallest components are undergoing almost continuous change or replacement.

System architects would then need a methodology to decide on the appropriate boundaries for the cells and to choose the correct change absorbers. System constructors and maintainers would then need tools that report on the cell structure and how it would react to particular changes.

This differs from much modular construction in that the cellular boundaries are not as rigid. In most modular systems, the modules are statically bound together and it is not possible to replace modules, or add modules without massive reconstruction.

It is expected that architects will deliberately build in significant change absorption capacity so as to avoid rigidity. This requires investment during design and construction, but it should be more than repaid during the system's life-time of change. Some computational costs are inevitably incurred as information and control passes across cell boundaries. The designer can trade these change costs by adjusting the size of cells and the units of work and information transmitted between them.

The repertoire of change absorption techniques can be categorised as:

1. automatic transformers;
2. partial transmitters; and
3. enquiry mechanisms.

[3]Not necessarily automatically, someone may have to redefine the view to maintain the old image.

6.1 Automatic Transformers

These attempt to absorb change entirely and to continue to transmit the same information. Typical examples are communication protocols, for example RPC mechanisms on top of TCP/IP or the database interfaces on PCs such as ODBC [MICROSOFT, 1993]. Another example is the object transmission mechanism of CORBA [Schaffert, 1992]. In such cases, the intermediate machinery will perform transformations to recover from representational changes across the interface.

An early example in the database context was the System/R access mechanism. Access code was compiled to deliver results for a query under the assumption that the database had a certain structure. If, when the query was eventually applied, it transpired that that structure had changed, the access code would be regenerated automatically to achieve the *same* result.

When considered from the viewpoint of absorbing change within a system we might expect significant developments in these standard interfaces. They effectively give guarantees to cells that they can rely on certain invariants. It is likely to become a major architectural issue, identifying precisely what invariants a cell can assume. For example, there may be some global invariants and others that are provided or promised more locally. Without such invariants it is almost impossible to construct a system. There is a challenge finding sufficient invariants without seriously inhibiting change.

It is probable that reflection [Stemple *et al.*, 1992] will be an important technology for building automatic change handlers. Just as type enquiry, program generation and reflection can be combined to give type dependent behaviour [Stemple *et al.*, 1990; Sheard, 1991] so it can use delivery and target descriptions to automatically generate transformers.

6.2 Partial Transmitters

The example already given of database views illustrates this kind of mechanism. It hides from the programs/users connected to the view: schema changes, changes to data values outside the view and some operations. It transmits all data changes that are within the view.

Another example is the use of subsumption at interfaces, such as in ADT signatures and as procedure parameters. This hides change which introduces additional properties and behaviours in excess of those specified. It transmits changes that reduce or change the properties of the expected type. It also transmits changes to the accessible values.

As was exhibited in the health care example, additions should not always be hidden. It should be possible to specify that at some interfaces any deviation from an exact match is unacceptable and requires change propagation.

However, it is possible to admit interfaces that propagate increases but perhaps hide certain removals from the object's properties. The increases might always be passed on to the target because its job was to present or preserve full information. However, established formats might be important so that omitted data should be replaced by a default (typically a null value) so that the code, etc. in the target does not have to be altered.

Combinations of subsumption and substitution would permit an interface to apparently present a constant type, absorbing all type changes with automatic transformations. Clearly the transformation functions would soon become complex and would need to be structured systematically. They would inevitably lose information and this is potentially dangerous. The change management tools that reported on the propagation structure would need to make it easy to obtain notification of their dangers.

It seems likely that an extensive repertoire of these partial transmitters will evolve as typical change propagation patterns are recognised.

6.3 Enquiry Mechanisms

The two previous mechanisms essentially consider the cells as passive objects. It is possible to make them active in the change process and to introduce special classes of cells that mediate in the change propagation process.

An important form of active change handling is that of *environment enquiry*. Here the cell changes its behaviour of its own volition on the basis of what it finds in its environment. Good examples appear on personal machines: word processors offer fonts based on the fonts they find on the system, and they offer printers after enquiring which are available. Similarly, program building aids and spreadsheets, enquire what libraries are mounted before presenting the user with the available options. In all cases it is expected that changes will occur, and that these will result in changed component behaviour.

Dynamic linking mechanisms and command interpreters find and load program based on information they find when they search for a module or command.

Such enquiries have been systematised in distributed file servers as name servers and in the ISA architecture for distributed object systems as object trading [ANSA, ; ISO,].

A common feature of these enquiry mechanisms is that a cell or cells (in our vocabulary) is introduced to hold some information that is expected to change. Programs causing such changes then record the information in these inquiry servers. Cells that wish to be sensitive to these changes then precede operations that use the affected facility by enquiries to the relevant enquiry server.

This use of enquiry mechanisms is important as it permits anticipated changes to be propagated completely automatically. In persistent object systems it is of course trivially easy to arrange that the data against which the enquiry is made is made persistent and changed transactionally. Thus, a cell having made an enquiry may trust the result of the enquiry until it has been used.

It is expected that such enquiry mechanisms will be used systematically to avoid having to explicitly propagate change.

7 Managing Program Change

Several authors have recently looked at the possibilities of managing program change in persistent object stores [Connor, 1991; Kirby, 1993; Cutts, 1993; Kirby *et al.*, 1992; Farkas *et al.*, 1992]. All three aspects of the above categorisation may be observed:

1. A standard structure in the persistent object store holds information about the available programs — thus the store acts as an enquiry server.

2. the program parts are held in variables which are themselves reached via bindings in **env** instances — which act as partial transmitters. Consequently, changes such as the insertion and removal of other program parts are hidden (see Section 3.3.2).

3. Changes to program parts that do not change their type are transmitted automatically whereas changes that do alter the part's type are detected and require programmers to manually accommodate the change.

The handling of some kinds of change to the types of program components might be automated with the caveats given above. This could be done by automatically introducing a "correcting procedure" that accepted the old call and called the new form, or vice-versa.

It has already been shown that in these systems it is easy to specify and control the propagation of changes to program components. For example, it is possible to statically bind to a value (using higher-order procedures as values and persistent identities in the source — hyper-references) immutably, so that whatever happens the value chosen at construction will be used. At the other extreme, programs can find other program parts they require every time they use them. As this all takes place in a transactional store, it is "safe" as programs using other code will behave like readers on that code.

It is expected that this structure of program parts will be regularised into various patterns of binding that allow the builders and architects to choose the corresponding patterns of change propagation. It is suspected that the necessary primitive types describing various forms of binding have already been developed in experimental systems. The patterns still need formal identification and capture in the form of defined type constructors.

8 Managing Type Change

Type change (or schema change) is perhaps the most difficult class of changes to manage in a PAS. There are several problem areas.

1. It is desirable to preserve as much of the type graph as possible unperturbed. Types refer to other types when they are defined, e.g. the parameters given to a constructor are referred to by the generated type. When type T1 is changed it is undesirable that every type that refers to T1 should appear to have changed — it would often produce a substantial cascade of changes. On the other hand, it is difficult to avoid this as for any model of type equivalence those types have changed.

2. Existing instances of the types must be changed. This can be done either in an immediate but expensive sweep of the store or incrementally whenever the objects whose type has been changed are next used.

3. The graph of instance references must be preserved but the preceding process may involve making new copies or changing addresses. Indirection mechanisms are one solution commonly employed in databases but they lose any performance potential of references.

4. The programs that handle the data either have to operate with the new data unchanged or respond to the changes.

Several partial solutions to this class of problem have already been discussed above. They are also well known in the literature [Skarra and Zdonik, 1987; Wegner and Zdonik, 1988;

Connor *et al.*, 1990b]. Solutions to all these problems can be arranged if the costs of anticipating change are tolerable. For example, objects that may change could be defined using the following constructor.

type ChangeProtected[InvariantPart] **is record**[fixed: InvariantPart; extra: **any**]

Any object is now declared with a type such as that in the following example.

type Patient **is** ChangeProtected[**record**[name: **string**; ...; next_of_kin: **string**]]

This works because, both at the type level and at the instance level the outer record structure is invariant and so type references, instance references and program interfaces are unperturbed by changes that are localised to the *extra* field. Initially all instances of this field would hold a null value.

When the previously described change occurs, a new structure for responsible-person information is introduced.

type ExtraPatient1 **is record**[responsible_person: **string**]

All subsequent and upgraded *Patient* objects now have a value of type *ExtraPatient1* injected into their *extra* field. This can be accessed via a projection by programs that need to use this data. All other programs are unchanged. The population of instances can be changed incrementally.

But there are some drawbacks to this method. An extra field access has been introduced and code has become more complex to write and execute. Optimisations and automatic code generation might overcome these problems. However, there is a more serious problem. The quality of the types as a description of the data has been seriously impaired. It is only in running text and in similarities of identifiers that the relationship between *Patient* and *ExtraPatient1* is established. The operational system and future programmers would not know that the field *extra* of *Patient* was only meant to have values of type *Extrapatient1* injected into it and no other.

Such notations will suffice for experiments that allow classes of change absorber to be investigated. Subsequently, more precise notations that actually limit the allowed changes to intended patterns, will be required. It is suspected that new primitives in the type system defining other classes of infinite union will be needed and/or defining other type matching rules. These will then be combined into standard patterns using defined type constructors.

9 Conclusions

The challenge of building Persistent Application Systems that will satisfy expectations is very great and of economic importance. They should enhance human cooperative behaviour. The work of the last decade on the development of persistence and object oriented systems has been much concerned to facilitate this design and construction process. The fundamentally important aspect of that research is to develop uniform computational environments so that unnecessary complexity in the Persistent Application System is avoided. There is still opportunity for further improvement.

It is apparent that a successful PAS will be expected to service its user community for a very long time, tens or hundreds of years. As the sophistication of PASs increases these required lifetimes will extend. In consequence an even greater challenge is to facilitate the maintenance and evolution of PASs. If that challenge is not met and the PASs become rigidly resistant to change or even just uneconomic to change then they will seriously harm the performance of their user communities. They will do this by inhibiting the normal processes of improvement that go on among a group of interacting people.

To achieve the required uniformity careful attention has been paid to orthogonality. For example, the rights to persistence are equal for values of all types and the parameters for type constructors are any type. Uniformity is also achieved by removing unnecessary duplication of concepts. The example given in the paper is the convergence of data model and type system.

An important issue, not covered in this paper, is how to maintain and evolve this uniform platform with adequate stability over the long lifetimes of PASs [Atkinson, 1992]. PASs built on technology for which that problem hasn't been solved — most persistent and object-oriented systems perhaps — will fail their user communities as they will not be able to capitalise on improved technology.

Type constructors enable the definition of new modelling constructs. This will allow the development of constructs appropriate to classes of PASs or even to parts of a PAS. Infinite union types are important in permitting code to be written which accommodates change. They have the property that new types can be injected into them and manipulated within them that were not defined when that manipulation code was written. Subsequently the injected type can be recovered by projection.

The preferred basis for accommodating change is an incremental approach to maintenance operations. This would extend throughout the life cycle. It is argued that only such an approach will ensure the capacity for change. It requires that from requirements capture, through design, construction and maintenance, the PAS is considered in terms of small units which are able to change with varying degrees of independence.

Although the above approach makes change possible it still leaves many technical problems to be solved. The maintainers still need accurate and relevant information to guide them as they plan and perform changes. Some work is reported which exploits the stable reference properties of persistent object systems to collect and maintain that information [Sjøberg, 1993b].

However, the various modes of change propagation between components have not yet been fully described. Consequently, the change management tools that can be built at present are of limited utility. In practice, PAS builders will use various techniques for limiting the propagation of change, but with present practices these are not identified and to tools are indistinguishable from other code.

It is proposed that forms of change propagation be identified. Here the interface components that permit various aspects of change to be absorbed or transmitted through a boundary have been called "change absorbers". The research community could therefore study the existing change propagation arrangements and requirements in order to properly identify an adequate repertoire of change absorbers. If these are then defined, named and provided by the technology they will enhance the capacity of PAS to change and permit the construction of tools that are more helpful when managing and implementing change.

The focus on describing the behaviour of the system under change will be the most important outcome of such a line of research. As indicated in the paper, many *ad hoc* developments to accommodate change and actively respond to it can be found in current research and current technology. The challenge is to systematise these into regular structures so that they can be more easily and extensively used by PAS architects, builders and maintainers. If this is done it will enhance the quality as well as the durability of a typical Persistent Application System.

Acknowledgements

Discussions with Tim King, Ray Welland, Richard Connor and Rich Cooper were important in formulating the ideas. In particular, Professor Morrison's team — including Richard Connor, Quintin Cutts and Graham Kirby — at St Andrews University, Scotland provided the language Napier88 that engenders thought about these ideas and allows them to be explored. We were much helped by Paul Philbrow in the preparation of this paper and some of the reported work was supported by a European Community ESPRIT Basic Research Action, $FIDE_2$, number 6309. Dag Sjøberg was also supported by the Research Council of Norway, Division NAVF.

References

[Abiteboul and Hull, 1987] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

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

[Albano *et al.*, 1993] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An introduction to the database programming language Fibonacci. Technical Report FIDE/93/64, ESPRIT Basic Research Action, Project Number 6309—$FIDE_2$, 1993. 30pp.

[ANSA,] The ANSA Model for Trading and Federation, AR.005, APM.

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

[Atkinson *et al.*, 1983] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983.

[Atkinson *et al.*, 1989] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Deductive and Object-Oriented Databases. Proceedings of the First International Conference on Deductive and Object-Oriented Databases (Kyoto, Japan, 4th–6th December 1989).* Elsevier Science Publisher B.V., 1989.

[Atkinson *et al.*, 1993] M.P. Atkinson, F. Matthes, and J.W. Schmidt. Progress with bulk types, 1993. Report in preparation, ESPRIT Basic Research Action, Project Number 6309—$FIDE_2$.

[Atkinson, 1978] M.P. Atkinson. Programming languages and databases. In S.B. Yao, editor, *The Fourth International Conference on Very Large Data Bases (Berlin, West Germany, September 1978)*, pages 408–419, September 1978.

[Atkinson, 1992] M.P. Atkinson. Persistent foundations for scalable multi-paradigmal systems. Invited paper. In M.T. Özsu, U. Dayal, and P. Valduriez, editors, *Pre-Proceedings of the International Workshop on Distributed Object Management (Edmonton, Canada, 18th-21st August 1992)*, 1992. The final proceedings will be published as: M.T. Özsu, U. Dayal and P. Valduriez (eds.), *Distributed Object Management*, Morgan Kaufmann, 1992.

[Bancilhon *et al.*, 1992] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, 1992.

[Berman, 1991] S. Berman. *P-Pascal: A Data-Oriented Persistent Programming Language*. PhD thesis, University of Cape Town, Department of Computer Science, August 1991.

[Birtwistle *et al.*, 1973] G.M. Birtwistle, O.J. Dahl, B. Myrhaug, and K. Nygaard. *Simula BEGIN*. Auerbach Press, Philadelphia, 1973.

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

[Cardelli, 1989] L. Cardelli. Typeful programming. Technical Report Digital Systems Research Center Report 45, Digital Eqipment Corp., Systems Research Centre, 130 Lytton Avenue, Palo Alto, Calif., USA, May 1989.

[Cardelli, 1990] L. Cardelli. The Quest language and system (tracking draft). Technical report, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, June 1990.

[Codd, 1979] E.F. Codd. Extending the relational model of data to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[Connor *et al.*, 1990a] R.C.H. Connor, A.L. Brown, Q. Cutts, A. Dearle, R. Morrison, and J. Rosenberg. Type equivalence checking in persistent object systems. In A. Dearle, G.M. Shaw, and S.B. Zdonik, editors, *Implementing Persistent Object Bases, Principles and Practice. Proceedings of the Fourth International Workshop on Persistent Object Systems, Their Design, Implementation and Use (Martha's Vineyard, USA, September 1990)*, pages 154–167. San Mateo, CA: Morgan Kaufmann Publishers, 1990.

[Connor *et al.*, 1990b] R.C.H. Connor, A. Dearle, R. Morrison, and A.L. Brown. Existentially quantified types as a database viewing mechanism. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Proceedings of the Second International Conference on Extending Database Technology (Venice, Italy, March 1990)*, number 416 in Lecture Notes in Computer Science, pages 301–315. Springer-Verlag, 1990.

[Connor, 1991] R.C.H. Connor. *Types and Polymorphism in Persistent Programming Systems*. PhD thesis, Department of Computational Science, University of St Andrews, 1991.

[Connor, 1993] R.C.H. Connor, 1993. In preparation: a survey paper on persistent type systems. Department of Computational Science, University of St Andrews.

[Cutts, 1993] Q.I. Cutts. *Delivering the Benefits of Persistence to System Construction and Execution*. PhD thesis, Department of Computational Science, University of St Andrews, 1993.

[Dearle and Brown, 1988] A. Dearle and A.L. Brown. Safe browsing in a strongly typed persistent environment. *The Computer Journal*, 31(3), 1988.

[Dearle, 1989] A. Dearle. Environments: A flexible binding mechanism to support system evolution. In B.H. Shriver, editor, *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, Volume II Software Track (January 1989)*, pages 46–45, 1989.

[Ellis and Stroustrup, 1990] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Farkas *et al.*, 1992] A. Farkas, A. Dearle, G.N.C. Kirby, Q.I. Cutts, R. Morrison, and R. Connor. Persistent program construction - a new paradigm. In A. Albano and R. Morrison, editors, *Fifth International Workshop on Persistent Object Systems. Design, Implementation and Use (San Miniato, Italy, 1st-4th September 1992)*, Workshops in Computing. Springer-Verlag in collaboration with the British Computer Society, 1992.

[Goldberg and Robson, 1983] A. Goldberg and D. Robson. *Smalltalk80: The Language and its Implementation*. Addison Wesley, 1983.

[Hammer and McLeod, 1981] M. Hammer and D. McLeod. Database description with sdm: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[Hoare, 1975] C.A.R. Hoare. Recursive Data Structures. *International Journal of Computer and Information Science*, 4(2):105–132, 1975.

[Hughes and Connolly, 1990] J.G. Hughes and M. Connolly. Data abstraction and transaction processing in the database programming language RAPP. In F. Bancilhon and O.P. Buneman, editors, *Advances in Database Programming Languages. Based on Proceedings of the Workshop on Database Programming Languages (Roscoff, Brittanny, France, September 1987)*, ACM Press, Frontier Series, chapter 11, pages 177–186. Addison-Wesley Publishing Company and ACM Press, 1990.

[ISO,] Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model, ISO/IEC JTC1/SC21/WG7, ISO.

[Kirby *et al.*, 1992] G. Kirby, R. Connor, Q. Cutts, A. Dearle, A. Farkas, and R. Morrison. Persistent hyper-programs. In A. Albano and R. Morrison, editors, *Fifth International Workshop on Persistent Object Systems. Design, Implementation and Use (San*

Miniato, Italy, 1st-4th September 1992), Workshops in Computing. Springer-Verlag in collaboration with the British Computer Society, 1992.

[Kirby, 1993] G.N.C. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems.* PhD thesis, Department of Computational Science, University of St Andrews, 1993.

[Koch *et al.*, 1983] J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, and C.A. Zehnder. Modula/R report, Lilith version. Technical report, Institute fur Informatik, Eidgenossische Technische Hochschule Zürich, February 1983.

[Matthes *et al.*, 1992] F. Matthes, A. Rudloff, J.W. Schmidt, and K. Subieta. The database programming language DBPL user and system manual. Technical Report FIDE/92/47, ESPRIT Basic Research Action, Project Number 3070—FIDE, 1992.

[Matthes, 1992] F. Matthes. *Generic Database Programming: A Linguistic and Architectural Framework.* PhD thesis, Fachbereich Informatik, Universität Hamburg, Germany, September 1992. (In German).

[MICROSOFT, 1993] Microsoft ODBC programmer's reference, June 1993. 01.03.0005.

[Morrison *et al.*, 1989] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. The Napier88 reference manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989.

[Morrison, 1979] R. Morrison. *On the development of algol.* PhD thesis, Department of Computational Science, University of St Andrews, 1979.

[Object Design Inc., 1991] Object Design Inc. *The ObjectStore Technical Overview.* Product Marketing, One New England Executive Park, Burlington, Mass, MA 01803, USA, May 1991.

[Ohori *et al.*, 1989] A. Ohori, O.P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proceedings of the ACM SIGMOD 1989 Conference on the Management of Data (Portland, Oregon, May-June), SIGMOD Record 18, 8, June 1989*, pages 46–57, 1989.

[Ontologic Inc., 1991] Ontologic Inc. *The ONTOS Developer's Guide.* Three Burlington Woods, Burlington, Mass, MA 01803, USA, 1991.

[Reiser, 1991] M. Reiser. *The Oberon System — User Guide and Programmer's Manual.* Addison-Wesley Publishing Company, Wokingham, 1991.

[Richardson and Carey, 1987] J.E. Richardson and M.J. Carey. Programming constructs for database system implementation in EXODUS. In *Proceedings of the ACM SIGMOD 1987 Conference on the Management of Data (San Francisco, CA, 27th-29th May)*, pages 208–219, 1987.

[Schaffert, 1992] C. Schaffert. CORBA: OMG's object request broker. In M.T. Özsu, U. Dayal, and P. Valduriez, editors, *Pre-Proceedings of the International Workshop on Distributed Object Management (Edmonton, Canada, 18th-21st August 1992)*, 1992. The final proceedings will be published as: M.T. Özsu, U. Dayal and P. Valduriez (eds.), *Distributed Object Management*, Morgan Kaufmann, 1992.

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

[Schwartz *et al.*, 1986] J.T. Schwartz, R.B.K. Dewar, E. Dubinski, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[Sheard, 1991] T. Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, 1991.

[Shipman, 1981] D.W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[Sjøberg *et al.*, 1993] D. Sjøberg, M.P. Atkinson, J. Lopes, and P.W. Trinder. Building an integrated persistent application — a multi-author, multilevel, Napier88 project. In *Proceedings of the Fourth International Workshop on Database Programming Languages (Manhattan, USA, 30th August–1st September 1993)*. Springer-Verlag, 1993. To appear.

[Sjøberg, 1993a] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, January 1993.

[Sjøberg, 1993b] D. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems*. PhD thesis, Submitted to University of Glasgow, July 1993.

[Skarra and Zdonik, 1987] A.H. Skarra and S.B. Zdonik. Type evolution in an object-oriented database. In B.S. Shriver and P. Wegner, editors, *Research Directions in Object Oriented Programming*, Computer Systems, pages 393–415. MIT Press, Cambridge, MA, 1987.

[Stemple *et al.*, 1990] D. Stemple, L. Fegaras, T. Sheard, and A. Socorro. Exceeding the limits of polymorphism in database programming languages. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Proceedings of the Second International Conference on Extending Database Technology (Venice, Italy, March 1990)*, number 416 in Lecture Notes in Computer Science, pages 269–285. Springer-Verlag, 1990.

[Stemple *et al.*, 1992] D. Stemple, R.B. Stanton, T. Sheard, P.C. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson, and S. Alagic. Type-safe linguistic reflection: A generator technology. Technical Report FIDE/92/49, ESPRIT Basic Research Action, Project Number 3070—FIDE, 1992. 29pp.

[Wegner and Zdonik, 1988] P. Wegner and S.B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-oriented Programming (Oslo, August 15-17, 1988)*, Lecture Notes in Computer Science 322, pages 55–77. Springer-Verlag, 1988.