A Tool for Introducing Persistent Programming

Lukito Nugroho
Department of Electrical Engineering
Gadjah Mada University
Jl. Grafika 2, Yogyakarta 55281
Indonesia
lukito@ugmgtw.ugm.ac.id

Geoff Sutcliffe
Department of Computer Science
James Cook University
Townsville
Australia, 4811
geoff@cs.jcu.edu.au

Abstract

Programming languages, using conventional file systems, are known to be ineffective at managing objects that outlive their creator programs. Researchers have shown that persistent programming, that is programming that involves persistent objects, can overcome the problem. Although persistent programming has proved to be advantageous and persistence systems designed for programmers are available, persistent programming is not yet widely practiced. This work develops a persistent programming tool for introducing programmers to persistent programming. The heart of the tool is a persistent object structure manager (POSM) that lets programmers create, use, and delete persistent object structures from inside a C program.

Key words Persistence, Persistent programming

CR categories D.1.m (Programming techniques: Miscellaneous), D.2.2 (Software engineering: Tools and techniques), D.3.3 (Programming languages: Language constructs and features).

1. Introduction

Many popular programming languages, such as Pascal and C, provide facilities to define various data structures in memory, which can be used to create *objects*. The objects are usually organised to form particular structures, called *object structures*. The example below shows how a data structure is used to create `Student` and `Subject` objects. Figure 1 shows how these objects are organised into an object structure. The `Subject` objects are organised into linked-list object structures, and each linked-list is attached to a `Student` object. The `Student` objects are organised as a binary tree object structure. This example shows that an object structure may be a part of a larger object structure. Conversely, an object structure may be divided into smaller object structures - the *substructures* of the larger object structure. An individual object is the smallest (trivial) object structure.

Example

```
typedef struct subject
    {
    char subject_code[6];
    char subject_name[10];
    struct subject *next;
    } Subject;
```

```
typedef struct student
    {
    int student_number;
    char student_name[20];
    Subject *subjects;
    struct student *left;
    struct student *right;
    } Student;
```

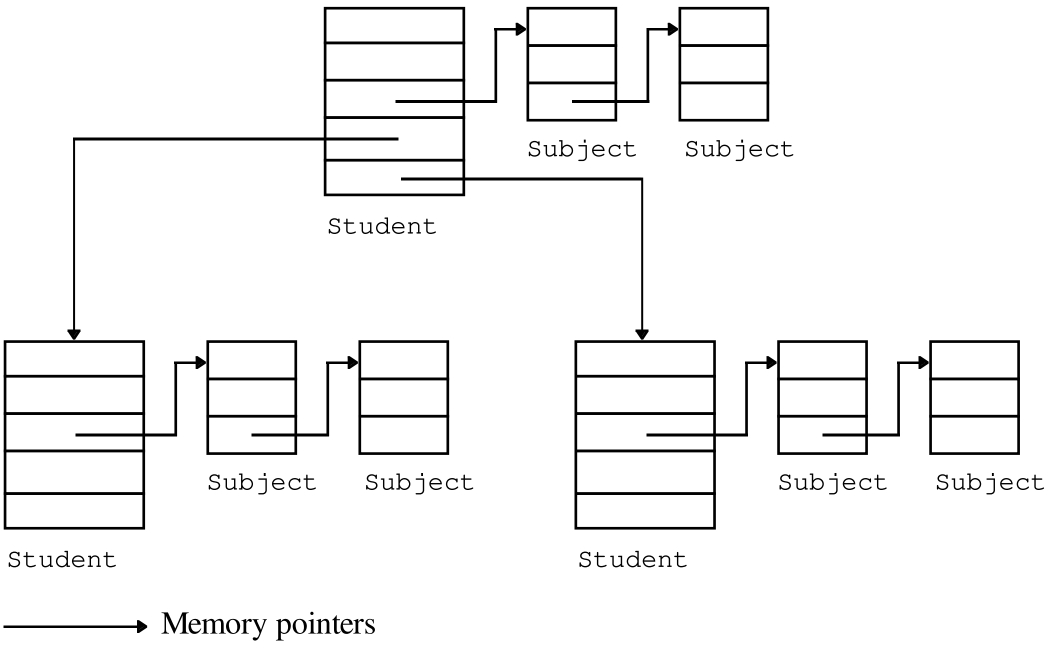

Figure 1 - The `Student` tree in memory

Managing *long-term* object structures (object structures that outlive the programs that create them) using conventional programming languages has certain disadvantages. The lack of control that programming languages have over object structures in disk files introduces overheads in programming activities (Atkinson, Bailey, Chisholm, Cockshott, & Morrison, 1983). At the higher level, bigger applications demand an abstraction to object structures higher than just at the physical level. DataBase Management Systems (DBMSs) are designed to solve this problem by offering their users a logical view of objects. However, even bigger and more complex applications cannot be handled properly using conventional DBMSs (Cattel, 1991). One of the approaches to solving this problem is to integrate programming and DBMS features. *Persistence* of object structures, that is the continued existence and useability of object structures (Atkinson, et al., 1983), becomes an important focus. The critical distinction between long-term object structures and persistent object structures is that long-term object structures simply store the data in the objects involved, while persistent object structures also retain the structural properties of the object structure (e.g., links between nodes in a binary tree).

Researchers have shown that persistent programming can be advantageous for programmers (Atkinson, et al., 1983), and several persistence systems have been designed for programmers (Atkinson, et al., 1983; Agrawal & Gehani, 1990; Bancilhon, 1988; Pinakis & McDonald, 1991; Richardson, Carey, & Schuh, 1993; Straw, Mellender, & Riegel, 1989). Support for persistent programming is also becoming available at the operating system level, e.g. the Grasshopper operating system (Dearle, di Bona, Farrow, Henskens, Lindström, Rosenberg, & Vaughn, 1994). However, persistent programming is not widely practiced. A hypothesised reason is programmers' inertia in their customs, and resistance to learning something new (Weinberger, 1971).

This research aims to build a programming tool to introduce persistent programming. Because of programmers' inertia and resistance, the tool has to be easy to learn and use. Based on this requirement, the tool is implemented as a functional extension of the C language, using C functions. The tool takes the form of a persistent object structure manager (POSM), which extends the C language with capabilities to manage persistent object structures. The extended C language is called **Persistent C** or **pC**.

2. The Design of pC

2.1. Basic ideas of providing persistence

An object structure is accessed in memory by initially pointing an access pointer to one of the member objects, called the *root* object. From the root, access is performed by traversing the *closure* of the object structure, that is the rest of the objects reachable from the root. Persistent C uses a related mechanism for providing persistence of object structures. It uses the *reachability* idea; if the root object persists, so do all objects that form the closure. This is similar to what PS-Algol (Atkinson, et al., 1983) does. Persistence of an object structure is achieved by copying it to the *persistent store* as disk files. The object structure's structure must be kept intact, so pointers must also be copied. Before being copied, pointers are *swizzled*, to have them point to the correct objects on disk (Moss, 1992). There will be no structural difference between the original object structure and its copy on disk.

Accessing persistent object structures in pC is aimed to be as easy and flexible as accessing memory object structures, i.e., pC adheres to the principle of persistence independence (Atkinson, et al., 1983). Two important aspects are identified. First, a programmer must be able to work with persistent object structures as easily as working with object structures in memory, including pointers. This is achieved in pC by providing facilities to easily convert object structures between their memory and persistent forms. Programmers then work with the object structures in memory as usual, and do not have to work directly with the persistent forms. In this aspect, pC is equivalent the ordinary C language, providing the familiarity of C's working environment to programmers. Second, programmers must be able to access persistent object substructures independently. This is similar to the access independence of object substructures in memory; given access to an object structure, a substructure can be accessed without affecting other substructures. Access independence also provides a basis for reusability and multiple user accesses to persistent object structures. pC provides the option to store substructures of an object structure separately in the persistent store. Each such substructure can be accessed independently, but at the same time the connections between the substructures are maintained. The granularity of storage is specified by the user (see Section 2.2). At the most fine grained level each individual object may be stored separately, providing fine grained access independence. If larger substructures are used then more compact storage can be achieved and access is faster because file I/O operations are reduced.

To implement the ideas mentioned above, the POSM provides several C functions as interfaces to the persistent object store. This implementation leaves programmers' working environment of C unchanged. Persistent C, through the POSM, manages accesses to persistent object structures.

Each disk file used by the POSM contains an indivisible object structure (possibly a single object), called a *unit* persistent object structure (unit-POS). Conversely, a unit-POS occupies a single file, called a *unit file*. With respect to the POSM, every disk file is a unit file and acts as the unit of storage on disk. Each unit-POS may be accessed independently in the persistent store, but the substructures of a unit-POS cannot be accessed independently. When a unit-POS is stored memory pointers are swizzled into *offset pointers*, which point to locations specified as offsets in the unit file. On the other hand, to maintain the connections between unit-POSs (to form larger object structures) disk pointers of another type, called *links*, are used. Links point to unit-POSs. Figure 2 illustrates how the object structure of Figure 1 can be made persistent, with each student's substructure being stored as a separate unit-POS in a unit file. The pointers within each student's unit-POS are converted to offset pointers, while the pointers between each `Student` are converted to links.

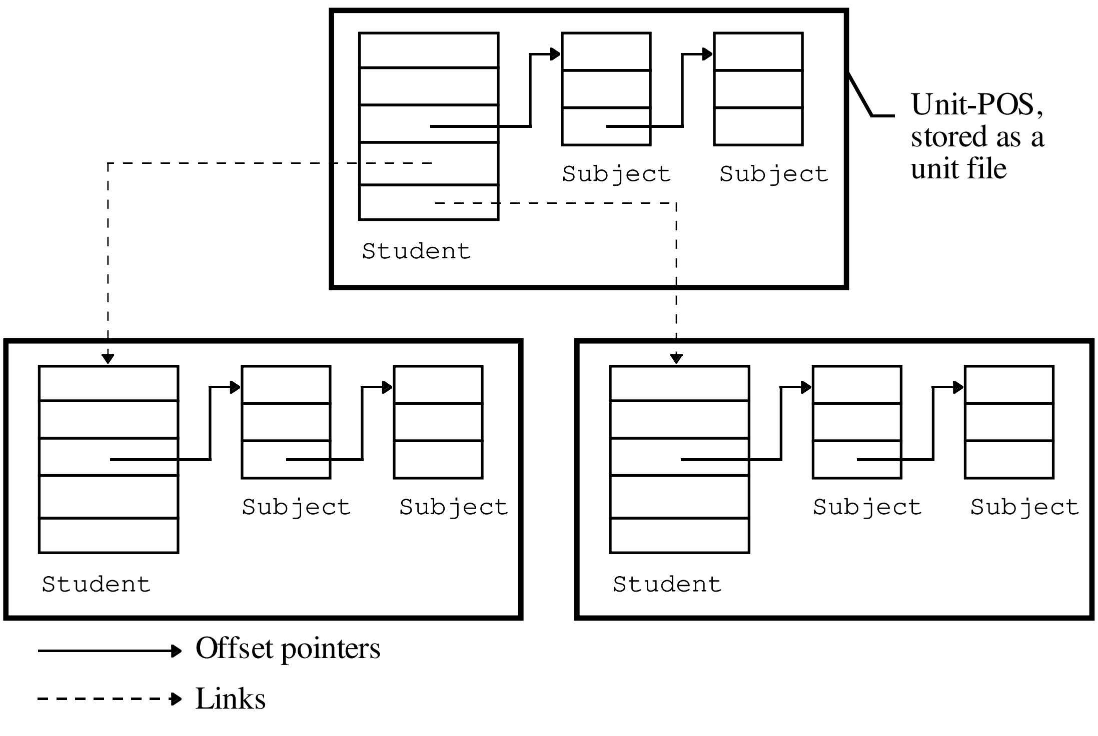

Figure 2 - The `Student` tree in the persistent store

The introduction of unit-POSs raises a need for unit-POS identification. An identifier, called a *UID* (for *U*nit-POS *ID*entifier), uniquely identifies each unit-POS in the persistent store. The UID is the logical handle of a unit-POS; links point to unit-POSs using their UIDs. However, it is not sufficient to have only UIDs as persistent object structure handles. Physical accesses are performed on unit files, which are identified by file names. There is a one-to-one relationship between UIDs and file names that must be recorded. A directory table is created for this purpose. The directory table lives in the persistent store as a unit-POS, and is central to the POSM; it records all necessary information about all unit-POSs.

2.2. Object structure copying

The POSM uses a depth-first method for copying object structures from memory to the persistent store and vice versa. The algorithm used is adopted from (Sajeev & Olszewski, 1993). For each object in the object structure, the POSM processes the fields[1] consecutively. Fields are traversed using their physical addresses, one after another. Because the address of a field is merely the field's offset from the beginning of the object that contains it, it is necessary to determine the preceding fields' sizes. Field size calculation requires type information, which is not available at run-time. Hence it has been necessary to develop an auxiliary typing mechanism.

A *typestring* is a string, derived from a data structure definition, that represents the type of an object (Nugroho, 1994). Serving as run-time type information, a typestring must always be specified when an object structure is to be made persistent or retrieved back into memory. The typestring of an object is stored in the directory table. When a persistent object structure is retrieved into memory, type checking is performed by comparing the specified typestring with the typestring in the directory table. The typestring for the object structure shown in Figure 2 is `"sia20c-*sa6c-a10c-*r-#r#r-"`. The substring `"sa6c-a10c-*r-"` corresponds to the `Subject` objects, with the `"*r"` specifying that the linked-list pointer must be stored as an offset pointer. The `"#r"` substrings specify that the left and right `Student` subtree pointers must be stored as links, so that each student's substructure is stored as a separate unit-POS.

From the typestring the sizes of the fields of an object can be calculated. Identifying that there are two groups of C compilers: those that perform data alignment and those that do not, field size calculation is performed in two steps. The first step computes the basic field sizes. For compilers that do not perform data alignment, this is sufficient. For the other type of compiler, a second step that adjusts field sizes to reflect data alignment is performed.

---

[1] Fields may be `struct` fields, elements of an array, or simply base data type variables.

3. The Persistent Object Structure Manager (POSM)

3.1. Some management aspects of the POSM

The POSM views the persistent store as a shared persistent memory. The ownership and access permissions of persistent object structures must be managed so that concurrent accesses may occur.

The POSM uses a password mechanism to identify the owner of a persistent object structure. When a user creates a new persistent object structure, the user supplies a password to establish ownership of the persistent object structure. The password is stored in the directory table. Password matching is performed at every access attempt.

As well as a password, the user also specifies the access mode of a new persistent object structure. The access mode specifies the right to access the persistent object structure. There are three access modes: *read-write*, *read-only*, and *none*. Together with the password, the access mode determines how a user other than the owner is permitted to access the persistent object structure. The owner of a persistent object structure always has read-write access.

The POSM provides a very simple mechanism to manage concurrent accesses of persistent object structures. Only one access at a time can be performed on a persistent object structure. A global access lock is placed at the beginning of an access, and is released at the end of it. A waiting mechanism is implemented to handle access contentions. It should be noted that the POSM does not provide access synchronisation. If write and read accesses are considered atomic, then programmers can build transactions that consist of a series of accesses. However, the lack of synchronisation may cause lost update and dirty read problems (Gray & Reuter, 1993).

3.2. The persistence functions

Recall that a user interacts with the persistent store using the POSM functions. In performing their tasks, the POSM functions rely on the persistence functions: `pprintf` and `pscanf`. These functions perform the basic activities of providing persistence, copying object structures from memory to disk and vice versa. The POSM functions are implemented on top of the persistence functions. Their syntax is as follows.

```
int pprintf (char *filename, char *typestring, void *object);

int pscanf (char *filename, char *typestring, void *object);
```

The `pprintf` function creates a unit-POS by copying an object structure from memory to disk, while `pscanf` does the reverse. The `filename` parameter names the unit file in which the unit-POS is stored, the `typestring` parameter shows the type of the unit-POS. For `pprintf` the `object` parameter specifies the memory address of the root of the object structure being made persistent. For `pscanf` the `object` parameter specifies the memory address where the root of the unit object structure is to be stored after being read from disk.

pprintf and pscanf share a similar design. They implement the same copying algorithm; only the direction is different. Another difference is in space allocation for objects. Having the advantage of compiler-supported memory space allocation, pscanf uses the malloc routine, while pprintf must do disk space allocation manually because there is no support from the C language for such a thing.

3.3. The POSM functions

The POSM provides a set of functions that enable a user to access persistent object structures.

1. The write_obj function.
   This function is an enhancement of the pprintf function. It writes a memory object structure to disk. If the object structure being written does not already exist in the persistent store, then write_obj creates a new persistent object structure. If the object structure already exists in the persistent store, then write_obj modifies the existing persistent version. As a consequence, it is not possible to overwrite an existing persistent object structure in a single step. To overwrite, the old persistent object structure must first be deleted (using delete_obj, see below), then the new one is written. This approach strives to achieve similarity to a write operation performed on a memory object structure. For example, a tree structure can be modified by adding or removing nodes, but the tree itself always exists[2]. On the other hand, if the tree is to be replaced by another structure, it has to be 'deleted' first (for example, by freeing the memory space allocation) before the new structure is created.

   The write_obj function has the following syntax.

   ```
   int write_obj (char *rootname, char *typestring, char *password,
                  char access_mode, void *object);
   ```

   The rootname parameter specifies the unit file where the root unit-POS resides. The password and access_mode parameters specify the password and access mode, respectively. The other two parameters are the same as those of pprintf.

2. The read_obj function.
   This function is used to read a persistent object structure into memory. The syntax is shown below. The parameters are the same as for write_obj.

   ```
   int read_obj (char *rootname, char *typestring, char *password);
   ```

3. The delete_obj function.
   This function removes a persistent object structure from the persistent store. In a deletion operation, the POSM must detect if any substructure of the object structure being deleted is also a substructure of another object structure. The unit-POSs in such substructures, called

[2]Even if all nodes are removed, the tree still exists conceptually, but not physically.

intersection unit-POSs, cannot be physically removed. Consider Figure 3. If the intersection unit-POSs are removed upon deletion of R1, then R2 becomes invalid.

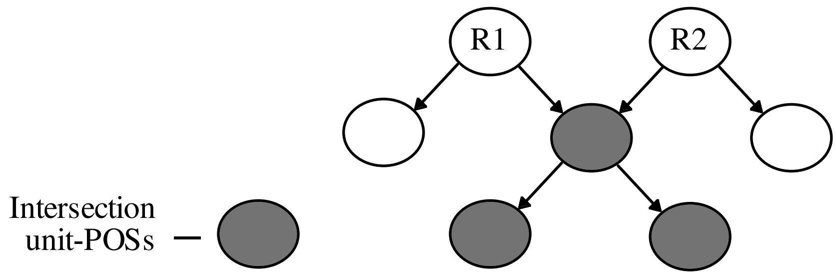

Figure 3 - Intersection unit-POSs

Instead of physically removing intersection unit-POSs, the deletion process simply unlinks them from the persistent object structure being deleted. In this manner they can still exist as parts of other persistent object structures. If a unit-POS is no longer part of any persistent object structure, and is part of an object structure being deleted, only then is it physically deleted from the persistent store. This mechanism is called *delayed deletion*.

The `delete_obj` function has the following syntax.

```
int delete_obj (char *rootname, char *password);
```

4. The browser.
   It is necessary for a user to be able to view the persistent store. Browsing the persistent store requires interpretation of the contents of unit files so that the data inside objects, including structural information represented by offset pointers and links, is easily understood. These requirements cannot be fulfilled by conventional file utilities, so a special browser is provided.

   The browser is called using the following syntax:

   ```
   browse [/options] [unit file name]
   ```

   Table 1 shows the available options for the browser. If no `options` are given, the fields of root unit-POSs are displayed. If no `unit file name` is given, all unit files are displayed.

Option	Meaning
`r`	Displays all fields of all unit-POSs in the persistent object structure rooted in `unit file name`.
`f`	Displays the file names of all unit-POSs in the persistent object structure rooted in `unit file name`.
`t:typestring`	Filters root unit-POSs by their typestring. All fields of all root unit-POSs that match the `typestring` are displayed.

Table 1 - Options for the browser

The browser is implemented as a standalone program. The POSM also provides a function, called `list_obj`, that can retrieve information about persistent object structures. This function can be used to query the persistent store from within user programs (indeed, this is simply what the browser does).

4. Discussion

The C language was chosen as the base language for the POSM because of its availability and popularity. Object-oriented languages, such as C++, are also becoming popular, but such a language was not chosen for two reasons. First, there are several existing persistence systems that are based on such languages (Richardson, et al., 1993; Agrawal & Gehani, 1990; Straw, et al., 1989). Second, it is known that not all programming problems can be handled elegantly using object-oriented techniques (Sommerville, 1992), so choosing a procedural language as the base for pC was appropriate.

One of the design concepts of pC was to make minimal change to the base language. This has been achieved by implementing the persistence features as C functions. This approach is different from that taken by E (Richardson & Carey, 1989) and O++ (Biliris, Dar, & Gehani, 1993), two persistence systems that are based on the C++ language. In E and O++ persistence features are achieved with help from other modules and (in E) by modifying the C++ compilation system. At the language level, both E and O++ also introduce new language constructs to provide persistence (i.e., `db` types and pointers in E and dual pointers in O++). Programmers need to expend extra effort to learn these new language constructs. Moreover, some of the new language constructs cannot be mixed perfectly with original constructs of the base language. By not introducing anything new at the language level, pC is easier to learn and use. Programmers work in an ordinary C programming environment. In memory, programmers work with object structures using their normal style of C programming. Persistent object structures are accessed through C function calls. The trade-off of the "minimal change" constraint imposed in pC is that it has not been possible to push pC to the limit, as in other systems. By using external modules, E and O++ can provide extra features that pC does not have. As an example, there is no type information for object structures available at run-time in C, so a separate type mechanism (typestrings) has had to be devised for pC. The use of typestrings, currently supplied manually by programmers, is currently a weakness of the pC system (however, it would be possible to preprocess programs to extract the typestrings, as done in Joyce-Linda (Pinakis & McDonald, 1991)).

To provide persistence of object structures, pC makes copies of memory object structures on disk. Similarly, to use a persistent object structure, a copy is made in memory. With this approach, programmers view the persistent store as a *white-box*. This term, borrowed from the area of software testing (Myers, 1979), suggests that programmers can 'see' the inside of the persistent store. The persistent store is not hidden or abstracted away from programmers. PS-Algol, with its directory mechanism, is another persistence system that uses the white-box approach. The white-box approach is in contrast to the *black-box* approach adopted by some persistence systems. The black-box approach 'hides' or abstracts the persistent store away from programmers (Richardson & Carey, 1989; Agrawal & Gehani, 1990; Bancilhon, et al., 1988; Andrews & Harris, 1987; Morrison, Baker, Connor, & Dearle, 1989). In these systems programmers do not (need to) know the storage scheme in the persistent store.

The white-box approach adopted by pC helps programmers to understand the concepts of persistence. Novice pC programmers see that persistent object structures are object structures, previously manipulated in memory, which 'freeze' on disk. The 'frozen' object structures can be retrieved at any time by any program. This concept is easy to understand because it is delivered in terms that are already known by programmers. For example, links and offset pointers are very similar to memory pointers. Also, modification and overwriting of persistent object structures are performed similarly to modification and overwriting of memory object structures. Finally, storing and retrieving object structures use the same mechanisms as storing and reading files.

Equipped with the abstraction of the persistent store, programmers can encourage themselves to use persistent data structures optimally, without having to worry about design compromises. This is an advantage of persistence brought to the program design phase. Programmers have a safe feeling about the object structures that they create, because they understand that whatever object structures they have, they can always make them persist on disk. Of course, this is also true for other persistence systems, but the pMusic experience (Nugroho & Sajeev, 1995) revealed that the feeling is strong when the programmer knows that what exists on disk is a copy of what they have created in memory. It is a psychological effect; a new concept is more convincing if it has open characteristics so that the users can see how it works. Of course, this argument cannot be applied to all programmers. What is important for some programmers is that the system can reduce their burden in programming tasks, and they do not bother to understand the way in which the system helps them. However, as pC is aimed as a tool for programmers to learn and experiment with programming with persistence, the white-box approach is more appropriate than the black-box approach. With the white-box approach, programmers understand the concept of persistence easier and are able to distinguish between persistent object structures and objects that are stored in conventional databases or file systems. Further, this approach blends well with the philosophy of the C language that programmers are familiar with.

5. Conclusion

The aim of designing a programming tool for managing persistent object structures has resulted in the pC language. The language is a strict superset of the C language, and the extension has the form of a persistent object structure manager, called the POSM. With the POSM, C programmers can create, use, delete, persistent object structures, or retrieve information from the directory table to be used to access persistent object structures. The underlying design philosophy of pC is simplicity. Simplicity makes pC easy to learn and use.

References

AGRAWAL R. and GEHANI N.H. (1990): Rationale for the design of persistence and query processing facilities in the database programming language O++, *Proceedings of the 2nd International Workshop on Database Programming Languages*, pp 25-40. Morgan Kauffman.

ANDREWS T. and HARRIS C. (1987): Combining Language and Database Advances in an Object-oriented Development Environment, *OOPSLA 87 Proceedings*, pp 430-440.

ATKINSON M.P., BAILEY P.J., CHISHOLM K.J., COCKSHOTT P.W., AND MORRISON R. (1983): An Approach to Persistent Programming, *The Computer Journal*, 26(4), pp 360-365.

BANCILHON F., et al. (1988): The Design and Implementation of O2, an Object-oriented Database System, Dittrich K.R. (Ed), *Advances in Object-Oriented Database Systems*, pp 1-22.

BILIRIS A., DAR S., AND GEHANI N.H. (1993): Making C++ Objects Persistent: the Hidden Pointers, *Software-Practice and Experience*, 23(12), pp 1285-1303.

CATTEL R.G.G. (1991): *Object Data Management : Object-oriented and Extended Relational Database Systems*, Addison-Wesley Publishing Co.

DEARLE A., DI BONA R., FARROW J., HENSKENS F., LINDSTRÖM A., ROSENBERG J., & VAUGHN F. (1994): Grasshopper:An Orthogonally Persisten Operating System, *Computing Systems*, 7(3), pp 289-312.

GRAY J. AND REUTER A. (1993): *Transaction Processing : Concepts and Techniques*, Morgan Kaufmann.

MORRISON R., BAKER C., CONNOR R.C.H., AND DEARLE A. (1989): *The Napier88 Reference Manual*, Technical Report PPRR-77-89, Universities of Glasgow and St. Andrews.

MOSS J.E.B. (1992): Working with Persistent Objects: to Swizzle or not to Swizzle. *IEEE Trans on Software Engineering*, 18(8), pp 657-673.

MYERS G.ZJ. (1979): *The Art of Software Testing*, Wiley.

NUGROHO L.E. (1994): *A Programmer's Tool for Managing Persistent Object Structures*. Masters thesis, James Cook University.

NUGROHO L.E. AND SAJEEV A. S. M. (1995): Persistence in Music Data Structures, *Proceedings of the ACM Symposium on Applied Computing*, pp 27-31.

PINAKIS J.N. AND MCDONALD C.S. (1991): The Inclusion of the Linda Tuple Space Operations in a Pascal-based Concurrent Language, *Proceedings of the 14th Australian Computer Science Conference*, pp 45.1-45.11

RICHARDSON J.E. AND CAREY M.J. (1989): Persistence in the E language: Issues and implementation, *Software-Practice and Experience*, 19(12), pp 1115-1150.

RICHARDSON J.E., CAREY M.J., AND SCHUH D.T. (1993): The Design of the E Programming Language, *ACM TOPLAS*, 15(3), pp 494-534.

SAJEEV A.S.M. AND OLSZEWSKI J. (1986): *Manipulation of data structures without pointers*, Technical Report 76, Department of Computer Science, Monash University.

SOMMERVILLE I. (1992) *Software Engineering*. Addison Wesley.

STRAW A., MELLENDER F., AND RIEGEL S. (1989): Object Management in a Persistent Smalltalk system, *Software-Practice and Experience*, 19(8), pp 719-737.

WEINBERGER G.M. (1971): *The Psychology of Computer Programming*, Van Nostrand Reinhold.