Adaptive Data Stores

R. Morrison, A. Dearle and C.D. Marlin

Department of Computational Science
University of St Andrews
North Haugh
St Andrews
Fife KY16 9SS
U.K.

Abstract

Adaptive data is characterised by its ability to react to changes in the environment. Such data frequently occurs in Artificial Intelligence applications where the knowledge base may alter dynamically to adapt to new stimuli. Such systems are usually written in typeless, dynamically bound languages. Here we describe the concept of persistence and show how it may also be used in conjunction with a strong type system employing flexible binding mechanisms to construct adaptive systems. This has the advantage of greater static checking with all of its attendant benefits, without losing flexibility.

Keywords and phrases: persistence, strong typing, dynamic binding, production systems, adaptive systems.

1. Introduction

A popular topic for speculation in recent times is how Artificial Intelligence (A.I.) techniques may be used in the management of large software projects [7]. For example, the searching of component libraries in version control and configuration management systems can become so computationally expensive as to lend itself to a heuristic-based search rather than an algorithmic one. This type of activity has led Software Engineers to discover the "treasures" of A.I. in trying to find solution to their own problems.

A direct consequence of this is that Software Engineers have begun to examine A.I. systems and have discovered that such systems are often constructed without using many of the techniques, tools and concepts that are well know to themselves. These techniques, tools and concepts include specification techniques and languages, prototyping methods, various programming paradigms, version control, configuration management and software reuse. All of these are valuable in the construction of algorithm-based systems and should be equally valid for heuristic-based systems.

We believe that the Software Engineering and A.I. communities have for too long underestimated the significance of each other's work, often dismissing some major advance in the other area as not applicable or as too immature for use in "real" applications. Historically this has been caused by the Software Engineering community concentrating its efforts in the area of Data Processing where the data structures are non-recursive and statically defined, and therefore not suitable for complex modelling. Even in more modern relational systems, the data structures are still first order and applications have to contort their model to fit into the relational view of the universe. A.I. systems, on the other hand, make use of highly recursive, dynamically adaptive data in their representation of knowledge. Large complex structures evolve during the lifetime of the system rather than being statically specified. Therefore, in general, A.I. applications require more flexibility in the form of dynamically growing data, dynamic binding and adaptive knowledge representation. It is often felt that Software Engineering cannot always help with this.

In this paper, we will use one Software Engineering programming paradigm, that of persistent programming, and show how it may be used to construct systems with adaptive data. Unlike traditional A.I. techniques, the paradigm uses a mixture of static and dynamic binding for safety and flexibility, a type system for complex modelling and security and a dynamically callable compiler to ensure the integrity of the system. We will use the persistent programming language Napier [13] in our examples and show how adaptive data stores may be constructed using its facilities.

2. Persistent Programming

Persistent programming is one method that may be used to control complexity when constructing large systems. Complexity must be controlled to allow the user or developer to concentrate on the application rather than the method of use or construction. The best known method of controlling complexity is abstraction, where the details of a particular level may be ignored when viewing the system from a higher level. In building our adaptive data stores, we wish to use two abstraction mechanisms: the persistence abstraction itself and the abstraction provided by a type system.

2.1 Persistence

Persistence abstracts over the length of time that data exists and is usable [1]. In essence, long term data is manipulated in the same manner as short term data. The translation of data between the forms required by the programming language and the forms required by the database management system or the filing system is provided automatically by the system. The persistence abstraction therefore provides the user with a uniform model of data over time, yielding an integration of the programming language and its environment.

There are three major advantages to the persistence abstraction. Firstly, it simplifies the use of data. In a traditional system, there are three models of data: the real world model, the program model and the database model. These three models and the mappings between them must be kept mutually consistent. This leads to an intellectual overhead on the part of the users and distracts them from the application instead making them concentrate on the complexity of the relationships between the models. Since this distraction is an artifact of the system, rather than inherent in the application, it may be removed. In doing this, the persistence abstraction provides the user with two models to control: the real world model and the persistent programming model, together with one mapping between them.

The second major advantage of the persistence abstraction is that there is less code to be written, maintained and executed. This has two benefits in terms of software costs [4]. In non-persistent environments, short term data has to be translated into a form suitable for a database management system or file system in order to store it for a long period; to use the data again, the inverse translation has to be performed. The complexity of this operation is especially clear where the data structures are cyclic – a common occurrence in A.I. In this case, a marking algorithm must be used to scan the structure prior to long term storage and the relationship between the data must be reconstructible after storage. Added to the translation code, we also require code to perform the transfer of data to and from the long term store.

It has been estimated that up to 30% of the code in an application is concerned with the above translation and movement of data. If it is performed by the underlying system, substantial savings may be made in writing, maintaining and executing individual applications.

The third major advantage of persistence is that any protection mechanism on data is available uniformly to all data. Thus, the type system of a programming language would be enforceable on all data and not lost across the unnecessary mapping between short and long term data stores.

Large systems require long-lived data since the attention span of humans is short in computational terms and hence they have a requirement to store their data between activities. We contend that persistent systems provide these data stores in the most suitable form for controlling complexity. Elsewhere, we have described the advantages of including first class procedures [2], processes [12] and graphics [10] as data objects in a persistent store. We have also demonstrated how persistent systems may be used as an ideal base for Software Engineering environments [9] and assert here that they made be used with equal success in the construction of A.I. systems.

2.2 Type Systems for Persistent Data

The second abstraction mechanism for controlling complexity in our system is that of type. Ideally, we would like a set of types and a type algebra, so that by a succession of operations of the algebra and the provision of parameters, we could define a data type equivalent to any data model or conceptual data model [3]. This is "the type alchemist's dream" and, as yet, has not been fully achieved.

Type systems provide two separate facilities that often interact. They are the ability to structure data in a regular form and the ability to protect data from accidental or deliberate misuse. Historically, the domination of the protection aspects of type over the modelling aspects has led developers of A.I. systems to regard type systems as too inflexible for their needs. We believe that it is now appropriate to alter this balance in order to demonstrate the modelling benefits of strong type systems.

In determining an appropriate type system, the designer is faced with a balance between flexibility and safety. The safety in the system is derived from being able to express (or even prove) something about the program before it runs (i.e. statically), in order to improve confidence in its correctness. This explains the wish by most language designers to employ static typing as one of the devices for static checking.

A second aspect of static checking is that the programs so checked are more efficient. By performing the checking statically, the need for dynamic checking is removed, making the run-time representation of the program execute faster and in less space.

Taken to the extreme, statically checked systems are not very useful for A.I. since they cannot accommodate change, the very aspect we frequently wish to model. The mechanism is, however, very safe.

The flexibility in a type system is determined by how late the checking can be delayed before it is enforced. In some systems, the type checking is performed dynamically [8] as the operator is applied to the operand. Reasoning about such systems is more difficult than statically checked systems but they are more flexible. We have already established elsewhere [3] that dynamic checking is necessary in providing the persistence abstraction over separately prepared program and data.

Type systems can be devised that offer a mixture of static and dynamic typing more suitable to the requirements of A.I. applications. Where appropriate, static checking may be used for safety and dynamic checking for flexibility. The mixture of static and dynamic objects will be determined by the nature of the particular A.I. application.

We will now introduce the Napier type system [13], which provides for both static and dynamic checking under user control. There are an infinite number of data types in Napier, defined recursively by the following rules:

1. The scalar data types are integer, real, boolean, string, pixel, picture, file and null.

2. The type image is the type of an object consisting of a rectangular matrix of pixels.

3. For any data type t, *t is the type of a vector with elements of type t.

4. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, **structure**(I_1:t_1,...,I_n:t_n) is the type of a structure with fields I_i and corresponding types t_i, for i = 1..n.

5. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, **variant**(I_1: t_1,...,I_n: t_n) is the type of a variant with options I_i and corresponding types t_i, for i = 1..n.

6. For any data types $t_1,...,t_n$ and t, **proc**($t_1,...,t_n$ -> t) is the type of a procedure with parameter types t_i, for i = 1..n and result type t. The type of a resultless procedure is **proc**($t_1,...,t_n$).

7. For any procedure type **proc**($t_1,\ldots,t_n$ -> t) and type identifiers $T_1,\ldots,T_m$, **proc** [$T_1,\ldots,T_m$] ($t_1,\ldots,t_n$ -> t) is the type **proc**($t_1,\ldots,t_n$ -> t) universally quantified by types $T_1,\ldots,T_m$. These are polymorphic procedures.

8. For identifiers $I_1,\ldots,I_n$ and types $t_1,\ldots,t_n$, **env** is the type of an environment with fields I_i and corresponding types t_i, for i = 1..n.

9. For any type identifiers $W_1,\ldots,W_m$, identifiers $I_1,\ldots,I_n$ and types $t_1,\ldots,t_n$, **abstype** [$W_1,\ldots,W_m$] (I_1: $t_1,\ldots,I_n$: t_n) is the type of an existentially quantified data type. These are abstract data types.

10. The type **any** is the infinite union of all types.

11. For any user-constructed data type t and type identifiers, $T_1,\ldots,T_n$, the type t[$T_1,\ldots,T_n$] is the type t parameterised by $T_1,\ldots,T_n$.

12. For identifiers $I_1,\ldots,I_n$ and procedure types $t_1,\ldots,t_n$, **process**(I_1: $t_1,\ldots,I_n$: t_n) is the type of a process with entries I_i and corresponding types t_i, for i = 1..n.

In addition to the above data types, there are a number of other objects in Napier to which it is convenient to give a type in order that the compiler may check their use for consistency.

13. Clauses which yield no value are of type void, the same as a resultless procedure.

The world of Napier data objects is defined by the closure of rules 1 and 2, under the recursive application of rules 3 to 13. The infinite unions env and any are used to support persistence, as well as being a general modelling technique; they are dynamically checked. For more information on environments, the reader is referred to [6]. The types picture, pixel and image are used for graphics.

In the next section, we will describe the more important aspects of this type system through illustrations. The essential element for our purposes is that there should be a high degree of abstraction. Thus, in the above type rules, vectors and structures are regarded as store abstractions over all data types, procedures as abstractions over expressions and statements, processes as abstractions over flow control, abstract data types as abstractions over declarations, and polymorphism and parameterisation as abstractions over type. Allowing such abstract forms in the object space enables the programmer to store them and to specialise them for reuse. This reduces the total amount of code required in any system, since we only have to write them once and may reuse them many times.

3. Persistent Programming for A.I.

The Napier type abstractions may be used in classical A.I. systems where knowledge is embodied in an evaluation function, planning controlled by something like a minimax procedure and some form of α–β pruning, and learning simulated by altering the evaluation procedure. In Napier, the above can be achieved by the following. Planning is performed by a minimax procedure operating on a tree type. Since the type of the tree is fixed, there is no advantage to dynamically typing it. α–β pruning is performed by using a variant data type to constitute the tree and overwriting the pruned nodes by nil values. The evaluation procedure operates on a set of weights and factors. It may be parameterised by a list of these allowing the learning process to alter their number and values.

The tree can be represented by the following type definition in Napier:

> **rec type** tree **is variant** (branch : branch_type ; tip : null)
> & branch_type **is structure** (leaf : int ; left, right : tree)

That is, a tree can be, recursively, a branch or a tip. Branches have leaves and left and right sub-trees. Pruning a particular branch entails changing the left or right sub-tree from a branch to a tip.

Notice that, although data structures of the type tree are dynamically varying in nature, all aspects of type checking in a program employing them can be performed statically. This static checking brings with it important benefits in terms of programmer productivity and the correctness of the resulting software.

Napier supports first class procedures. That is, procedures are allowed the same civil rights as any other data object: they can be assigned and are allowed as the results of expressions, as parameters to and results of other procedures, and as elements of vectors, records and other data structures. The procedures are therefore higher order. A consequence of the procedures being assignable in the above discussion is that learning can be performed by over-writing the evaluation function by assignment. The new procedure will have encapsulated within it new weights, factors and so on.

The second kind of classical A.I. application, typified by Winograd's SHRDLU system [14], requires a procedural representation of data. The first class procedures of Napier may also be used here. The knowledge is represented in the system by procedures working on known facts. The representation of meaning and the rules of use of the system are also represented by procedures which may encapsulate, or be parameterised by, knowledge. Since the procedures may be freely assigned and manipulated, they can be substituted for one another in the system, allowing for very flexible composition of knowledge.

For production systems, where knowledge is represented by rules, we can also use Napier procedures. The knowledge base is of the form:

$C_1 \Rightarrow A_1$
$C_2 \Rightarrow A_2$
.
.
.
$C_n \Rightarrow A_n$

where the C_i are the conditions and the A_i are the corresponding actions. In a Napier data structure to represent this knowledge base, the C_i may be represented by boolean procedures whose definitions are stored in the data structure.

An example of a such data structure to represent a production system can be written in Napier as

```
rec type production_list is variant (production : production_type ; tip : null)
& production_type is structure (condition      : proc ( -> bool) ;
                                action         : proc () ;
                                next           : production_list)
```

This type represents a list of productions where each element is either a tip or a production. A production consists of two procedures – a condition which takes no parameters and results in a boolean value and an action which has a void procedure type – and the next element in the list.

The following is a recursive Napier procedure to scan the list sequentially and fire the first action which has an associated true condition:

```
let evaluator =  proc (productions : production_list)
                 if productions isnt tip do
                      if productions'condition ()  then productions'action ()
                                                   else evaluator (productions'next)
```

To run the production system, the evaluator procedure is repeatedly called with the production list as the actual parameter, as in

```
while not_finished do evaluator (list_of_productions)
```

We will now explore two extensions to this production system mechanism. The first is to develop the type of the conditions and actions so that they may take parameters. The second is to introduce non-determinism in the production evaluation.

In the above example, all the actions A_i have their environment enclosed within the procedure. That is, identifier occurences within the A_i are bound to particular values within the closure of the procedure. The same is also true of the conditions C_i. We wish to extend the above implementation so that both the conditions and the actions can be arbitrarily and dynamically parameterised. Before we can discuss how this may be achieved, however, we must briefly describe the polymorphic typing facilities of Napier.

Polymorphism is a mechanism whereby we can abstract over type. It is perhaps best explained by an example; in Napier

> **let** int_id = **proc** (x : int -> int) ; x

defines the identity procedure for integers. We may also define one for reals, such as

> **let** real_id = **proc** (x : real -> real) ; x

Polymorphism allows us to combine the two definitions by abstracting over the type, creating a generic identity procedure for example. This may be performed by

> **let** id = **proc** [t](x : t -> t) ; x

which is the identity procedure for all types; in fact, it has the type ∀t.t -> t . To call the procedure, we may write

> id[int](3)

which will return the value 3. The important point about this this form of type polymorphism is that it is all statically checkable. Objects of these types may be kept in the persistent store and then used to generate specialised procedures. This is an important method of software reuse [11].

Another form of type polymorphism employing dynamic checking occurs with the type any, which is the union of all types. Values may be injected into and projected from the type dynamically. An example of the polymorphic identity procedure for type any is

> **let** id = **proc**(x : any -> any) ; x

the type of which is **proc** (any -> any). To call this procedure, the parameter must be of type any. This can be done by injection, as in the following case:

> **let** this_three = id (any (3))

To retrieve the integer value from the object, we must project. An example of projection is the following:

```
project this_three as Y onto
    int  : Y + Y
    .
    .
    default : ...
```

To return to the production system example, we may conclude that the conditions C_i and the actions A_i in the production system may be coded with a single parameter of type any for dynamic checking. The production list could now be represented by the following type definition.

```
        rec type production_list is variant (production : production_type ; tip : null)
        & production_type is structure (            condition        : proc (any -> bool) ;
                                        action      : proc (any) ;
                                        next        : production_list)
```

This allows the environment to be packaged by the evaluation procedure into a suitable object and passed to the condition and action procedures. The procedures themselves retain a standard type and the system may be evaluated by an evaluation procedure similar to the one given earlier.

A final problem with such typing occurs when the environment packaging, and the later use within the procedures, is written without knowledge of each other. Since it is not known how the object is packaged, it cannot be unpackaged. For this, we require the browser technology described in the next section.

The second problem with production systems is in controlling the sequencing of the actions. So far, we have evaluated the conditions deterministically. By modelling the production system as a sequential process executing in parallel, we can use the **select** clause to introduce non-deterministic evaluation to our system.

We will define an evaluation type to be a process with two entries, one to cause an evaluation of the rules and one to shut down the process. It may be defined by:

```
        type evaluation is process (evaluate : proc () ; quit : proc ())
```

External to the process, the entries behave like the procedure types they are defined to have. Thus, the act of calling an entry is equivalent to calling a procedure. We will now define another procedure that will yield different instances of the process when called.

```
let evaluator =          proc ( -> evaluation)
                    evaluation with
                    begin
                         let done := false
                         while ~done do
                         begin
                            select
                               C1 ()  : receive evaluate () do A1 ()
                               C2 ()  : receive evaluate () do A2 ()
                               ...
                               Cn ()  : receive evaluate () do An ()
                                      : receive quit () do done := true
                            selected
                         end
                    end
```

The body of the procedure yields a process of type evaluation. The process loops, executing the **select** clause, until "done" becomes true. All the conditions C_i ,which must be of type boolean, are evaluated in order on each execution of the loop. An option in the **select** clause is said to be open if it does not contain a boolean expression or the boolean expression is true; otherwise it is closed. One of the open options is chosen for execution non-deterministically, subject to the constraint that if the clause to be executed is a **receive** clause, it will only be chosen if the entry can be received immediately. If none of the options can be immediately executed, e.g. if there is no entry pending, the process waits until one can be.

In the above, the conditions C_i constitute boolean guards. Any of the options can be chosen non-deterministically, and the appropriate action A_i executed, if the process has received a call to the evaluate entry. When it receives a quit entry call, it may terminate.

To create an evaluator process operating in parallel, we simply call the evaluator procedure as in the following:

```
let this_evaluator = evaluator ()
```

There may be many evaluators in the system. To initiate the evaluation of a production system, we must call the evaluate entry in the appropriate evaluator process. In the case of the process just created, this would take the form

```
this_evaluator (evaluate) ()
```

We have now established two separate ways of firing the conditions: deterministically or non-deterministically.

One final aspect of production systems which must be dealt with is that the number of conditions and actions may change dynamically. This is easily accommodated in our first, deterministic, version but when the conditions C_i and the actions A_i are represented by program rather than a data structure an even more powerful binding mechanism is required. For example, in the process type above, the number of C_i and A_i are fixed statically in the text of the process. An appropriate binding mechanism is described in the next section.

4. Data Evolution

Since the uses of data cannot be predicted, it is necessary to provide mechanisms that will support the evolution of data. In the previous section, we discussed ways in which systems could be constructed by encoding flexibility into the procedures that manipulate data. The limitation of this approach is that we cannot perform any action in the system that was not envisaged when the system was designed and written. Thus, even though Napier provides a rich set of binding mechanisms, there are some activities that are forbidden to ensure the integrity of the type-secure store.

One forbidden activity is type coercion. That is, the system does not allow the types of objects to be arbitrarily changed. Thus, for example, if a string representing the definition of a procedure is to be converted into a procedure, some special mechanism must be made available to check the validity of the conversion. Such a mechanism is called a compiler, which must be available at run-time within the persistent environment if the above objective is to be met.

The Lisp eval function [8] provides such a facility. We believe that some of the attraction of Lisp for A.I. programming is that it allows programs to be constructed, compiled and executed dynamically. This is necessary for handling adaptive data, such as programs that produce programs. However, the Lisp environment is entirely dynamic and typeless, which makes it difficult to employ modern Software Engineering techniques in the construction of A.I. systems using Lisp.

Apart from the facilities already described, the Napier system also provides a callable compiler. It is specified by the declaration

let compiler = **proc** (source : string -> any)

Thus, it is given a string representing the source and it returns a compiled procedure injected into the type any. To obtain the actual procedure, we must project out of the union. The compiler guarantees the type integrity of the system and is the only program allowed to perform these coercions. We say that it contains the "type magic" for the system. By utilizing the callable compiler, we can produce compiled code for new activities, for example to add new productions in the example of the previous section. This compiled code gives us the efficiency that we require and avoids having to have intermediate levels of interpretation on the data to achieve the same functionality.

We can utilize the callable compiler to implement adaptive data. This is done by a technique that we call *browser technology* [5]. A browser, in this sense, contains a knowledge base of actions that are selected by certain conditions becoming true. It is, of course, itself a production system.

A browser scans the persistent store to find objects to browse over, such as the memory associated with either of our implementations of a production system in the previous section. When it encounters an object, the browser interrogates its rule base to see if it has an action for that object. In the most primitive case, the interrogation of the rule base can be a matter of matching on structural type and the action would be the application of the known browser procedure to that object, allowing the inspection of its contents.

If, however, the browser does not find a rule for the object, it must produce one. This can be done by automatically generating the source of a browsing procedure for that type of object. The source is then compiled by the callable compiler and the resultant procedure, keyed by the type, is added to the rule base. The browsing can now continue. Since the browser program itself is data, by virtue of the provision of first class procedures, we have outlined a programming paradigm for adaptive data.

Thus, a browser might be used to inspect the list of productions if we were to use our deterministic method of implementing production systems. Alternatively, if we were to employ the non-deterministic method, it might well be that the evaluator procedure and the procedures for the conditions C_i and actions A_i are stored in the persistent store, in which case the same browser technology can be used to examine them.

Apart from allowing programs to inspect objects in the persistent store, in the manner of an interactive debugging aid [5], the same browser technology can be used to write programs which perform other actions based on the structure of data encountered in the persistent store. For example, all rule based production systems that alter the rule base can be programmed by this method. The evaluation process could be generated automatically from a list of conditions and actions.This allows the construction of non-deterministic searching of variable number of conditions and actions in a rule base. The process text is generated for the particular list, compiled by the compiler and activated by the controlling system automatically. Thus we have an automatic method of implementation for production systems, with non-deterministic evaluation and a variable number of conditions and actions within a type secure store.

The browser technology can also solve the problem, mentioned in the previous section, of packaging and unpackaging objects of type any in isolation. To unpackage, it is sufficient to obtain the type of the object from it and then automatically generate a program to unpack it.

5. Conclusions

We have outlined how some Software Engineering techniques may be used in the development of A.I. systems. Such use offers the same advantages as these techniques have demonstrated elsewhere, namely reduced development costs and increased reliability.

Two related modern Software Engineering paradigms were explored in some detail: persistence as an approach to the consistent treatment of short and long term data, and a related strong typing system. We have shown that this combination of paradigms can meet the demands of A.I. programmers for a flexible approach to the management of data and for the ability to treat program code as data, whilst still enjoying the advantages of the Software Engineering techniques mentioned above.

A particular example, that of a production system, has been described in sufficient detail to show how it can be programmed in a language such as Napier, which supports persistence and strong typing, without losing any of the flexibility normally obtained through the use of typeless languages such as Lisp. The same techniques used in this example can be employed in other A.I. programming situations requiring adaptive data.

In this way, A.I. systems may be constructed using Software Engineering techniques which have been used successfully in other application areas. Their use in the development of A.I. systems will bring the same significant benefits that it has done in these other areas.

6. Acknowledgements

This work was undertaken at the University of St Andrews during the study leave period of Dr. C. D. Marlin of the University of Adelaide. The work was supported by SERC grant number GR/E 75395. We are also grateful for the helpful comments of the referees.

7. References

1. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. "An approach to persistent programming". *Computer Journal*, **26**, 4 (November 1983), pp. 360-365.

2. Atkinson, M.P. & Morrison, R. "Procedures as persistent data objects". *ACM. TOPLAS*, **7**, 4 (October 1985), pp. 539-559.

3. Atkinson, M.P., Morrison, R. & Pratten, G.D. "Designing a persistent information space architecture". *Proc. 10th IFIP World Congress*, Dublin (September 1986), pp. 115-120.

4. Boehm, B.W. "Understanding and controlling software costs". *Proc. 10th IFIP World Congress*, Dublin (September 1986), North-Holland, Amsterdam, pp. 703-714.

5. Dearle, A. & Brown, A.L. "Safe browsing in a strongly typed persistent environment". *Computer Journal*, to appear (1988).

6. Dearle, A. "Environments: a flexible binding mechanism to support system evolution". *Proc. HICSS-22*, Hawaii (January 1989).

7. Jacquart, R.
"Software engineering and knowledge engineering: mutual relations".
Proc. 10th IFIP World Congress, Dublin (September 1986), North-Holland, Amsterdam, pp. 725.

8. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. & Levin, M.I.
The Lisp Programmers Manual. MIT Press, Cambridge, Massachusetts (1962).

9. Morrison, R., Bailey, P.J., Brown, A.L., Dearle, A. & Atkinson, M.P.
"A persistent store as an enabling technology for an integrated project support environment". *Proc. IEEE 8th International Conference on Software Engineering*, London (August 1985), pp. 166-172.

10. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.
"An integrated graphics programming environment". *Computer Graphics Forum*, **5**, 2 (March 1986), pp. 147-158.

11. Morrison, R., Brown, A.L., Carrick, R., Connor, R.C., Dearle, A. & Atkinson, M.P.
"Polymorphism, persistence and software reuse in a strongly typed object-oriented environment". *Software Engineering Journal* (December 1987).

12. Morrison, R., Brown, A.L., Carrick, R., Connor, R.C. & Dearle, A.
"On the integration of object-oriented and process-oriented computation in persistent Environments". *Proc. 2nd International Workshop on Object-Oriented Database Systems*, West Germany (1988).

13. Morrison, R., Brown, A.L., Carrick, R., Connor, R.C. & Dearle, A.
"The Napier Reference Manual". University of St Andrews, St Andrews, Scotland (1988).

14. Winograd, T.
Understanding Natural Language. Academic Press, New York (1972).