This paper should be referenced as:

Morrison, R., Connor, R.C.H., Cutts, Q.I. & Kirby, G.N.C. “Persistent Possibilities for Software Environments”. In **The Intersection between Databases and Software Engineering**, IEEE Computer Society Press (1994) pp 78-87.

Persistent Possibilities for Software Environments

R. Morrison, R.C.H. Connor, Q.I. Cutts & G.N.C. Kirby

Department of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland

Abstract

Persistent programming systems are generally recognised as the appropriate technology for the construction and maintenance of large, long-lived object based application systems such as software environments. Here some of the advantages of providing all the support required for the software process within a persistent object store with referential integrity are examined. It is shown that powerful new techniques may be supported by having system specifications, programs, configuration management tools and documentation all within a single persistent environment.

1 Introduction

In recent years considerable research has been devoted to the investigation of the concept of persistence and its application in the integration of database systems and programming languages [Atk78, ABC+83]. The benefits of orthogonal persistence have been described extensively in the literature [ACC82, ABC+84, AM85, AMP86, AB87, Dea87, MBC+87, Wai87, AM88, Dea88, Bro89, MBC+89, Con90, MBC+90]. These can be summarised as

- improving programming productivity from simpler semantics;
- removing ad hoc arrangements for data translation and long term data storage; and
- providing protection mechanisms over the whole environment.

Recently persistent programming systems have been developed that allow the complete software process to take place entirely within the persistent environment [BOP+89, MBC+89]. Thus each component of the software process can take advantage of the persistent environment. This paper focuses on one particular advantage, that of the referential integrity of the links (references) in the persistent store. The referential integrity of a link means that once a link to an object in the persistent environment has been established, the object will remain accessible for as long as the link exists. This can be used to effect in the following areas:

- construction and editing of programs;
- compilation of programs;
- linking of programs;
- execution of programs;
- configuration of applications from component programs;
- versioning of application components; and
- documentation of application components.

For example, naming schemes that operate by convention can be replaced by links to persistent objects. Such linking extends to inter-component relationships and allows the

software components such as programs, configurations, versions and documents to be cross-referenced by immutable links. The advantage is that the overall software process is more reliable since the conventions used to name and define associations among objects are replaced by links whose integrity is guaranteed by the persistent system.

When symbolic names are replaced by links, the representations of the software components are non-flat. In the context of documentation, this leads to hyper-text [Con87, Nie90], where documents contain links to other documents. In the context of programs it leads to the concept of the hyper-program [KCC+92], where program representations contain direct links to data objects of any type including other program components.

This paper illustrates how the use of links to persistent objects gives rise to new techniques for: programs, version control, configuration management and documentation. By using the same technique, similar advantages accrue to the other components of the software process which are not discussed here.

2 New Paradigms for the Software Process

2.1 Hyper-programming

In persistent systems, programs may be constructed and stored in the same environment as that in which they are executed. Objects accessed by a program may already be available in that environment at the time that the program is composed and thus links to the objects can be included in the program instead of textual descriptions. A program containing both text and links to objects is called a *hyper-program*.

Figure 1 shows an example of a hyper-program. The first link is to a first class procedure value which when called writes a prompt to the user. The program then calls another procedure to read in a name, and then finds an address corresponding to the name. This is done by calling a lookup procedure which is one of the components of a table package linked into the hyper-program. The address is then written out. Note that code objects (*readString* and *writeString*) are treated in exactly the same way as data objects (the table).

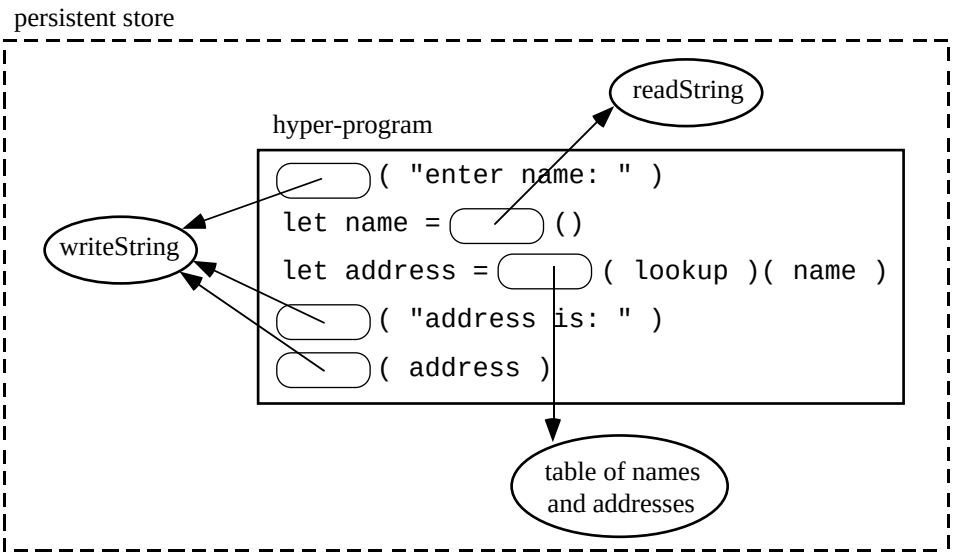

Figure 1. A hyper-program

The benefits of hyper-programming are discussed fully in [Kir92].

The major pay-off comes when the hyper-program itself is considered as an object. It may then contain links, perhaps hidden to the user, to any compiled or executable form. It is equally simple to arrange that the compiled or executable forms contain reverse links to the hyper-program. Thus the source object, the compiled object and the executable object may be kept in lock-step by a mechanism that is enforced by the referential integrity of the persistent system.

2.2 Hyper-code

One of the advantages of hyper-programming is the ability to use the hyper-program representation for both source, as above, and run-time representations of programs.

At run-time the hyper-program may be used to represent an active computation. This is possible due to the non-flat nature of the hyper-program representation. Free values in objects and procedures may be represented as links and the inherent sharing of values and locations referred to by links is preserved. This is not possible with textual representations of programs since the sharing is lost. The use of hyper-program source representations allows browsing and debugging tools to display meaningful representations of procedure closures, showing both source code and direct links to other components. This aids software re-use since documentation in the form of the original source code and documentation text—see later—can be made available for every procedure value in the persistent environment.

More importantly the unification of program representation leads to the possibility of a conceptual simplification of the programming activity. The *hyper-code* abstraction is one such conceptual simplification. It is a development of hyper-programming where only one representation of a program is required to be understood by the programmer. In constructing the program the programmer writes hyper-code. During execution or when a run time error occurs the programmer is presented with, and only sees, the hyper-code representation. Thus the programmer need never know about entities that the system may support for efficiency only, such as object code, executable code, compilers and linkers.

2.3 Version Control

Different styles of version control are provided in different systems. One possible mechanism using links will now be described.

The mechanism involves the concept of a *version controller*, which is self contained and solely responsible for the organisation of the versions of a particular object. Initially an object is registered with a version controller at the time of creation of the controller. Thereafter copies of versions may be checked out of the version controller and later checked in again after having being edited, thus creating new versions.

Hyper-code, version controllers, configurations and documentation are all software components and may therefore be versioned. The definition is recursive in that hyper-code may have links to version controllers; version controllers may provide versioning of other version controllers; configurations may be versioned; configurations may have links to hyper-code and version controllers, etc.

Each version controller presents two interfaces:

- one interface to the application builder, who specifies the initial object to be versioned and causes the evolution of new versions from existing versions; and
- another interface to the user of the version controller, who is only allowed to access the versions.

A version controller is used to give an abstract view of a software component, providing a logical grouping of its various concrete versions. There is also a need for abstract views of the versions within a version controller, so that the user of the version controller may specify which version is required without knowing about the data structures that allow navigation between versions. This is provided by an access path mechanism known as a *version window* that allows versions to be specified logically rather than with reference to the temporal order in which they were created [Rei89].

The version controller provides a number of version windows. Each version window views (is mapped to) a particular version and this mapping is controlled by the version controller. Thus to access a versioned object the user links to a version window which corresponds to a version.

The mapping from version window to version may be frozen, i.e. constant, or may change through time to provide access to different versions as the versions evolve. Figure 2 shows the structure of a version controller, its windows and its versions.

In this example the version window called *release2.0* is frozen and bound to a particular fixed version: the version for release 2.0 presumably. The windows *latestReliable3.4.2* and *latestExperimental4.2*, however, provide access to different versions as the system evolves. Note that the names of the concrete versions, *v0.0* etc, are visible only to the application builder. This is an example of a name space being layered on top of the linking graph for convenience (of the application builder).

Changing the mapping between a version window and its corresponding version may only be performed by the application builder, through the protected interface. Change strategies may be automatically provided or programmed but must always preserve type safety.

The ability to create new windows also allows the application builder to provide ‘frozen’ version windows. This is done by copying an existing, movable, version window—here movable means that the mapping from window to version may be changed—and specifying that its mapping is frozen and cannot be changed. Users of this window will now always access the same version. In contrast, users of a movable window will access a new version on the first access after the mapping is changed.

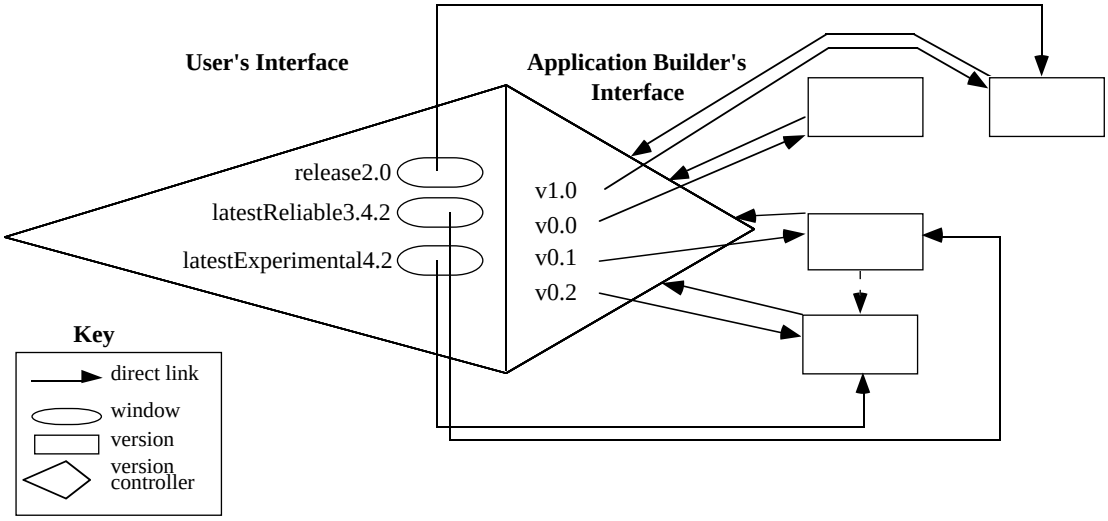

Figure 2. A version controller

The version controller gives an abstract view of the versions of a component. A good analogy for version windows is that of snapshots and views in query languages

[ABC+76, SWK+76, Pow85]. The snapshots are analogous to fixed version windows and the views are analogous to movable version windows that provide different versions as the system evolves.

2.4 Configuration Management

The presence of the persistent environment also allows re-evaluation of application configuration management.

There are two different kinds of configuration to deal with in a configuration management system. One is the logical configuration of an application which refers to the components used in the application; the other is the physical configuration which concerns the particular versions of each component. Both the logical and physical configurations must be recoverable from the system. Again, the term component refers to all entities in the software process, not just source code.

Persistent applications are built from locations, un-versioned values and version controllers. Since version controllers and configurations are values, the definition is recursive. Within a configuration the specification of the components may be by name or by link and in this respect the configurations are similar to hyper-code.

The configuration technique is to develop applications from a *target configuration* which is a logical view of the components of the application and their inter-relationships. The components are subsequently developed. The target configuration describes all of the components of the application whereas the components only contain links to the components that they use directly. These links can be used to discover the actual configuration of a component by inspecting the transitive closure of its hyper-code. The actual and target configurations, which may have diverged as the component evolved, can now be compared.

The first step in constructing an application is to specify the target configuration, which may be constructed graphically. The target configuration is purely a guide to the proposed configuration of the application and it enforces no restrictions on the actual construction. The target configuration can be constructed from existing values or it may contain representations of proposed components. For example, Figure 3 shows the target configuration for a simplified compiler. Diamonds represent version controllers and rectangles represent un-versioned values. Shaded objects signify objects that do not yet exist and unshaded objects are links to existing objects. The arrows represent *intended* links only: they do not represent any actual links between components. The compiler uses the standard procedures *readString* and *writeString*, these are un-versioned and exist already. The other major components are intended to be under version control, and either do not yet exist or links to them from the diagram have not yet been established. In this example the type checker module exists but the other do not.

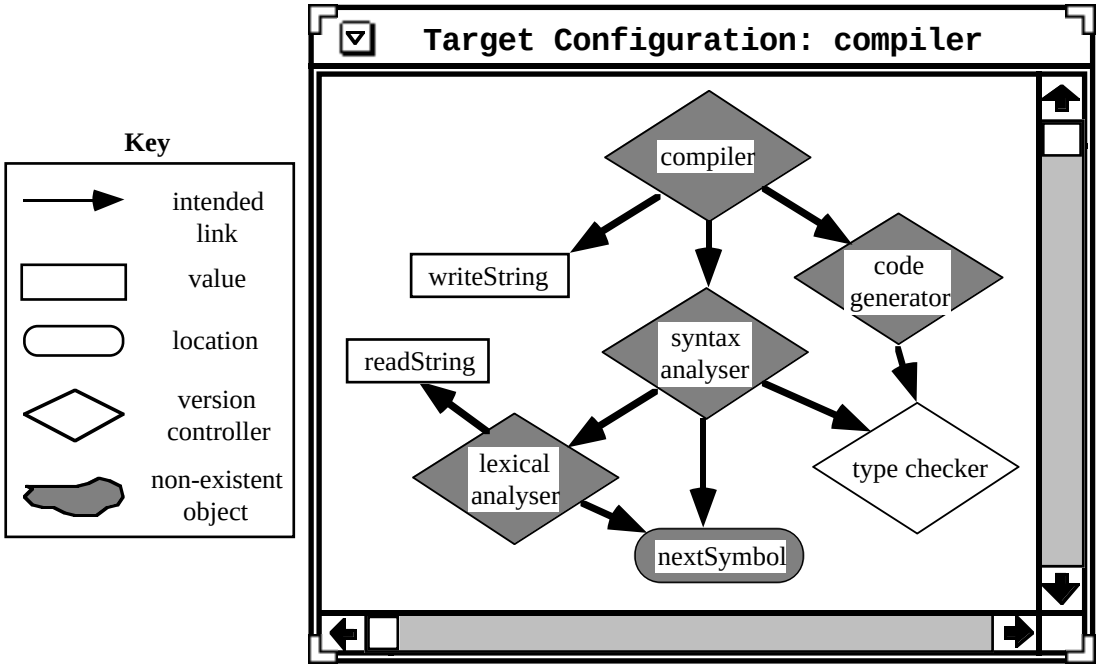

Figure 3. Compiler target configuration

The target configuration itself may now be placed under version control to cater for refinements to the design. When the compiler component is created it will contain a link to the version controller of the target configuration, which may then be used as a guide for further evolution of the compiler. Thus configuration management and version control information, hyper-code and applications may be kept in lock-step with each other.

Once the design is created, the hyper-code is written. It may contain text, links to version controllers, and links to un-versioned values. Figure 4 illustrates the hyper-code for the compiler.

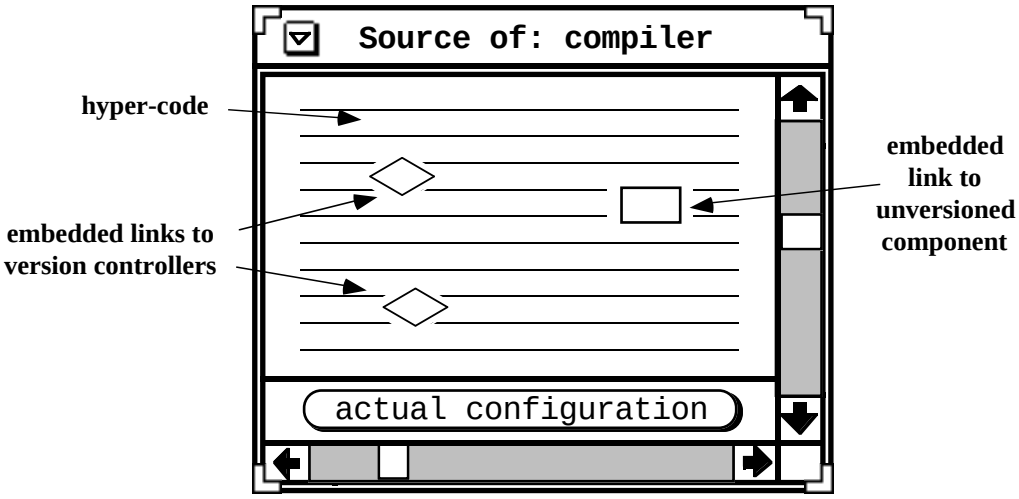

Figure 4. The compiler source

The window has a button to examine the actual configuration. This may then be compared with the target configuration, either manually or by the system. Notice that even for a single version of an application its configuration may change through time.

Figure 5 illustrates the stage of development where the type checker and lexical analyser have not yet been constructed.

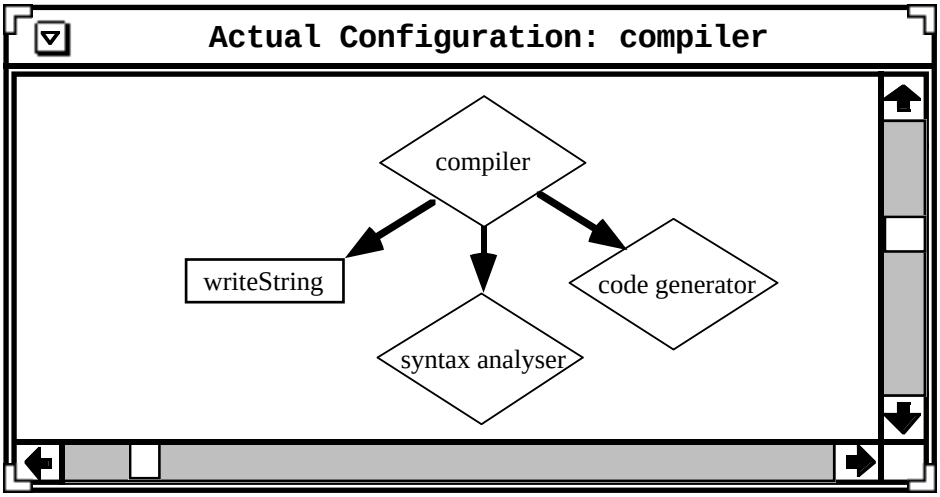

Figure 5. The compiler actual configuration

The novelty of the actual and target configuration approaches is that by using the links, real rather than perceived configurations can be discovered automatically. This allows the checking that is inevitable in evolving systems to be performed. Secondly, since all the system components may now be placed under version control, generic configurations may be constructed from which families may evolve.

2.5 Documentation

One of the most problematic aspects of system documentation is to ensure that it is consistent with the application that it is supposed to describe. Traditionally keeping the documentation with the application is done by association. By using the method described, these associations can be replaced by links and the relationship between application and documentation enforced by referential integrity.

Documents may contain links to objects such as the target configuration or hyper-code. In turn the documents are considered as objects and links to the documents can be placed in the target configurations and hyper-code. Figure 6 illustrates such a scheme.

The links between documents, version controllers, hyper-code and configuration information ensure that all are kept in lock-step and consistent with one another. This does not ensure that documentation is accurate, since that requires semantic interpretation, but does avoid the possibility of accidental loss while promoting documents to first class entities.

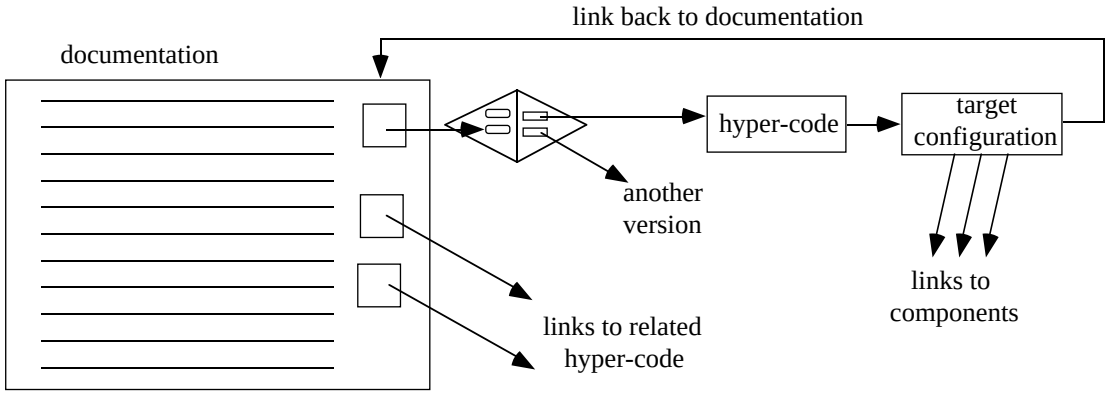

Figure 6. System documentation with links

3 Conclusions

Orthogonally persistent environments are by definition strongly typed, highly structured, and enforce referential integrity. File systems are traditionally composed of weakly typed, weakly structured components, and do not enforce referential integrity. The advantages to the software process described here all rely upon these differences in the objects manipulated by the program editors, compilers, linkers, version controllers and configuration managers. Thus hyper-programs are possible only because the typed links in the programs are guaranteed to be maintained during and after manipulation by an editor. Hyper-code is possible only because the compiler can cause source and executable versions of the same code to be reliably linked to each other, thus enabling a unified view of the program. The version control and configuration management strategy outlined is possible only because links placed in versions of code and data by the version controllers can be reliably interpreted to discover the dynamically changing configuration of a component.

The combination of these new concepts yields a software engineering environment in which a programmer need understand only the programming task. Hyper-programming removes any complexity introduced by an explicit linking mechanism; hyper-code removes the unnecessary conceptual gap between source and executable code, and the version control mechanism avoids the description of complex configuration information by allowing configuration details to be discovered as well as imposed.

4 References

[AB87] Atkinson, M.P. & Buneman, O.P. “Types and Persistence in Database Programming Languages”. ACM Computing Surveys 19, 2 (1987) pp 105-190.

[ABC+76] Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., Eswaran, K.P., Gray, J.N., Griffiths, W.F., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W., Putzolu, G.R., Traiger, I.L., Waid, B.W. & Watson, V. “System R: A Relational Approach to Database Management”. ACM Transactions on Database Systems 1, 2 (1976) pp 97-137.

[ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”. Computer Journal 26, 4 (1983) pp 360-365.

[ABC+84] Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. “Progress with Persistent Programming”. Universities of Glasgow and St Andrews Technical Report PPRR-8-84 (1984).

[ACC82] Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. “PS-algol: An Algol with a Persistent Heap”. ACM SIGPLAN Notices 17, 7 (1982) pp 24-31.

[AM85] Atkinson, M.P. & Morrison, R. “Procedures as Persistent Data Objects”. ACM Transactions on Programming Languages and Systems 7, 4 (1985) pp 539-559.

[AM88] Atkinson, M.P. & Morrison, R. “Types, Bindings and Parameters in a Persistent Environment”. In **Data Types and Persistence**, Atkinson, M.P., Buneman, O.P. & Morrison, R. (ed), Springer-Verlag (1988) pp 3-20.

[AMP86] Atkinson, M.P., Morrison, R. & Pratten, G.D. “A Persistent Information Space Architecture”. In Proc. 9th Australian Computing Science Conference, Australia (1986).

[Atk78] Atkinson, M.P. “Programming Languages and Databases”. In Proc. 4th IEEE International Conference on Very Large Databases (1978) pp 408-419.

[BOP+89] Bretl, B., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E.H., Williams, M. & Maier, D. “The GemStone Data Management System”. In **Object-Oriented Concepts, Applications, and Databases**, Kim, W. & Lochovsky, F. (ed), Morgan-Kaufman (1989).

[Bro89] Brown, A.L. “Persistent Object Stores”. Ph.D. Thesis, University of St Andrews (1989).

[Con87] Conklin, J. “Hypertext: A Survey and Introduction”. IEEE Computer 20, 9 (1987) pp 17-41.

[Con90] Connor, R.C.H. “Types and Polymorphism in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1990).

[Dea87] Dearle, A. “Constructing Compilers in a Persistent Environment”. In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1987).

[Dea88] Dearle, A. “On the Construction of Persistent Programming Environments”. Ph.D. Thesis, University of St Andrews (1988).

[KCC+92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag (1992) pp 86-106.

[Kir92] Kirby, G.N.C. “Reflection and Hyper-Programming in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1992).

[MBC+87] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “Polymorphism, Persistence and Software Reuse in a Strongly Typed Object-Oriented Environment”. Software Engineering Journal, December (1987) pp 199-204.

[MBC+89] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “The Napier88 Reference Manual”. University of St Andrews Technical Report PPRR-77-89 (1989).

[MBC+90] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Dearle, A., Rosenberg, J. & Stemple, D. “Protection in Persistent Object Systems”. In **Security and Persistence**, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag (1990) pp 48-66.

[Nie90] Nielsen, J. **Hypertext and Hypermedia**. Academic Press, New York (1990).

[Pow85] Powell, M.S. “Adding Programming Facilities to an Abstract Data Store”. In Proc. 1st International Workshop on Persistent Object Systems, Appin, Scotland (1985) pp 139-160.

[Rei89] Reichenberger, C. “Orthogonal Version Management”. In Proc. 2nd International Workshop on Software Configuration Management, Princeton, New Jersey (1989) pp 137-140.

[SWK+76] Stonebraker, M., Wong, E., Kreps, P. & Held, G. “The Design and Implementation of INGRES”. ACM Transactions on Database Systems 1, 3 (1976) pp 189-222.

[Wai87] Wai, F. “Distribution and Persistence”. In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1987) pp 207-225.