This paper should be referenced as:

Morrison, R., Connor, R.C.H., Cutts, Q.I., Dunstan, V.S. & Kirby, G.N.C. “Exploiting Persistent Linkage in Software Engineering Environments”. Computer Journal 38, 1 (1995) pp 1-16.

Exploiting Persistent Linkage in Software Engineering Environments

Morrison, R., Connor, R.C.H., Cutts, Q.I., Dunstan, V.S. and Kirby, G.

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

Abstract. Persistent programming systems are designed to provide technology for the construction and maintenance of large, long-lived object-based application systems. Many successful prototypes have been constructed and a large body of application building experience is emerging. Three common attributes of persistent systems are persistent linkage, strong typing and the referential integrity of data. Persistent linkage allows persistent objects to be included in the binding process. Strong typing guarantees that objects are only manipulated in a manner consistent with their type system descriptions. Referential integrity ensures that once a link (reference) to an object is established, its identity is unique and it persists over time. As a consequence no object can be deleted while another refers to it. Here we examine some of the advantages of providing software engineering environments within a persistent object system with strong typing and referential integrity. It is shown how the integration of system specifications, programs, configuration management tools and documentation all within a single persistent environment leads to powerful new techniques. This new power is achieved by sharing structured persistent data across the hitherto enclosing boundaries of system components.

Keywords. persistence, software engineering environments, links, strong typing, referential integrity, hyper-text, hyper-programming

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 (Atkinson, 1978; Atkinson, Bailey *et al.*, 1983). As a result a number of persistent systems have been developed including Abstract Data Store (Powell, 1985), Amber (Cardelli, 1985), Fibonacci (Albano, Bergamini *et al.*, 1993), Flex (Currie, 1985), Galileo (Albano, Cardelli *et al.*, 1985), Napier88 (Morrison, Brown *et al.*, 1994), PS-algol (PS-algol, 1988), TI Persistent Memory System (Thatte, 1986), Trellis/Owl (Schaffert, Cooper *et al.*, 1985) and Tycoon (Matthes and Schmidt, 1992). In each of these systems persistence is used to

abstract over the physical properties of data such as where it is kept, how long it is kept and in what form it is kept, thereby simplifying the task of programming. The benefits of orthogonal persistence have been described extensively in the literature (Atkinson, Chisholm *et al.*, 1982; Atkinson, Bailey *et al.*, 1984; Atkinson and Morrison, 1985; Atkinson, Morrison *et al.*, 1986; Atkinson and Buneman, 1987; Dearle, 1987; Morrison, Brown *et al.*, 1987; Wai, 1987; Atkinson and Morrison, 1988; Dearle, 1988; Brown, 1989; Connor, 1990; Cooper, 1990a; Cooper, 1990b; Morrison, Brown *et al.*, 1990). These can be summarised as:

- improving programming productivity as a consequence of simpler semantics;
- avoiding *ad hoc* arrangements for data translation and long term data storage; and
- providing protection mechanisms over the whole computational environment.

The persistence abstraction is designed to provide an underlying technology for long-lived, concurrently accessed and potentially large bodies of data and programs. Typical examples of such systems are CAD/CAM systems, office automation, CASE tools and software engineering environments (Morrison, Bailey *et al.*, 1985; Morrison, Brown *et al.*, 1987). Others include Object-Oriented Database Systems such as GemStone (Bretl, Otis *et al.*, 1989) and O_2 (Bancilhon, Barbedette *et al.*, 1988), which have at their core a persistent object store, and process modelling systems, which use a persistent base to preserve their modelling activities over execution sessions (Bruynooghe, Parker *et al.*, 1991; Curtis, Kellner *et al.*, 1992; Han and Welsh, 1993).

Some persistent programming systems allow software engineering environments to be completely supported within the persistent system (Currie, 1985; Bretl, Otis *et al.*, 1989; Morrison, Brown *et al.*, 1994). Thus each software environment component or activity, including process modelling, can take advantage of the persistent system. This paper focuses on the combination of three particular advantages, those of persistent linkage, strong typing and referential integrity. Although other integrated programming environments have been developed (Teitelbaum and Reps, 1981; Reiss, 1984; Teitelman and Masinter, 1984; Sweet, 1985; Habermann and Notkin, 1986; Dowson, 1987; O'Brien, Halbert *et al.*, 1987; Akima and Ooi, 1989; Bott, 1989; Thomas, 1989), the authors do not know of any that use persistent linkage, strong typing and referential integrity to gain the benefits described here.

The hypothesis presented in this paper is that persistent linkage (which allows persistent objects to be included in the binding process, thereby increasing the range of binding times), strong typing (which ensures type security), and referential integrity (which preserves identity) may be used in combination to improve both the construction and use of software engineering environments.

To illustrate the power of the combination of persistent linkage, strong typing and referential integrity in a persistent environment the following examples will be used:

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

This paper is concerned with a style of programming environment which is to a large extent closed: it is either single-language or it places a stringent discipline of usage on the systems to which it may be interfaced. This runs contrary to much current work on open systems and inter-operability. The purpose of the paper is to highlight what may be gained by imposing the discipline of persistent linkage, strong typing and referential integrity on a programming system.

2 Persistent Linkage

Binding mechanisms present the user with a trade-off between safety and flexibility (Atkinson, Buneman *et al.*, 1988; Morrison, Brown *et al.*, 1988; Morrison, Brown *et al.*, 1990). Dynamic binding is the most flexible as the binding is delayed until the latest possible time at which a choice can be made. In contrast, static binding is safer, in that static checking may be used to eliminate run-time binding errors. The programmer has to choose the mechanism most suitable for a particular application from the range of binding available within the construction system. Persistent systems present the user with the possibility of using persistent objects during the construction of an application. This increases the range of binding times over traditional systems. The focus of interest here is the exploitation of this increased binding range within software engineering environments.

2.1 Referential integrity

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 via that link for as long as the link exists. Furthermore, the identities of the objects are unique, and comparison of identity yields the same result independently of when it is performed. In a strongly typed persistent environment this also means that the type correctness of all such links is maintained, i.e. once a link has been established the type of the object linked to will not change.

In systems with explicit deletion, an object is physically deleted only when the last link to the object is removed (cf Unix hard links). Thus although the object may not be available for new links to be made to it, all extant links to it remain valid. In systems without explicit deletion, garbage collection may be used to determine when an object may be finally removed.

Using links with referential integrity can improve the safety of a system. Instead of referring to objects by some naming convention, anonymous links can be used. Once a link to an object is obtained, access to the object is guaranteed for as long as the link exists. For example, the configuration management tool *make* (Feldman, 1979) and the version control tools SCCS (Rochkind, 1975) and RCS (Tichy, 1985) all rely on a name space to identify the components of an application. If the conventions of the name space are improperly used the tool will fail; such improper use includes changing the name of a file from a different context. This possibility could be prevented if the tools used links instead of names. For example, the Vesta configuration management system achieves this by implementing its own file system and ensuring that the file descriptors have the referential integrity property (Chiu and Levin, 1993; Levin and McJones, 1993).

A further advantage of using links instead of names is that since access to objects is independent of the naming scheme, any number of naming schemes (including zero) may be layered on top of the linking graph for user convenience. As stated earlier, object access mechanisms involve a trade-off between safety and flexibility. Where names are replaced by links to give greater safety, flexibility is reduced since decisions about which particular objects to access are taken earlier. In applications where such flexibility is required it may be provided through user naming schemes.

When naming conventions are replaced by links, the representations of the software components become non-flat. This is because the representations themselves contain links to other components rather than symbolic names for them. In the context of documentation, for example, this leads to hyper-text (Conklin, 1987; Nielsen, 1990), where documents contain links to other documents. In the context of programs it leads to the concept of the hyper-program (Kirby, Connor *et al.*, 1992), where program representations contain links to data objects of any type, including other program components.

2.2 Persistent type systems

Type systems are normally viewed as providing two aids to the programmer: a modelling framework to aid the task of data abstraction, and a protection mechanism which prevents this modelling framework from being improperly used by a program. The most significant difference between persistent and non-persistent programming languages is that in a persistent language the long-term data is type secure. This results in a major shift in the emphasis of type system protection, from one of a safety mechanism over programs to that of a safety mechanism over the entire software system, including both programs and data (Atkinson, Buneman *et al.*, 1988; Connor, 1990; Morrison, Brown *et al.*, 1990). The assumption is made here that the persistent programming language supports higher-order procedures. This allows both code and data to persist in the same object store, subject to the same modes of use.

A persistent type system models protection over data which escapes from or originates outside the context of a program's text. In a non-persistent language the typechecker is usually invoked during the compilation of each program to check the consistent use of data modelling. Any data accessed externally, for example from a file or database system, is explicitly converted into the type system framework. In a persistent system, however, the program text may contain expressions which access external values in persistent storage. Before any program statement which uses such data is executed, a check must have been made that it will not violate any type system constraints placed on the data at the time it was created. A programming system that enforces such checking is termed *strongly typed*.

If type system constraints are checked dynamically there is no extra typechecking problem. When values are created their type system attributes are associated with them in such a way that they may be dynamically accessed whenever a check is required.

Dynamically checked type systems however lead to unreliable code as type errors are not detected until execution. In static type systems, type errors may always be detected before the execution of a program has commenced. Static type systems are well known to be achievable in non-persistent languages; the challenge is to achieve static typechecking within a persistent system. This seems at first to be an intractable challenge, as persistent programs require the ability to access typed data which is external to the program context. Recent research however has demonstrated an approach to achieving purely static typechecking within a fully integrated persistent programming environment (Connor, Atkinson *et al.*, 1993; Connor, Cutts *et al.*, 1995).

2.3 Persistent linkage

Programming languages support a number of different mechanisms for establishing persistent links from programs to values, locations and types. The degrees of freedom include linking to L-values (locations) or R-values (immutable values), and the time at which the linking takes place (Morrison, Brown *et al.*, 1990). The focus here is on the range of linking times, for which some possibilities are: during program composition, during compilation, during a separate linking phase, and during execution. However it should be remembered that a link can be made to an R-value and will always refer to that value, or it can be made to a location (L-value), in which case the value within the location can be changed to give the effect of dynamic binding.

Figure 1 illustrates the traditional method of using persistent data in a file system or database. Programs are prepared independently of the data and linked to the data during program execution. The programs contain the names or pathnames of the objects that they require. These are translated into object references for the duration of the program execution. There is also usually a separate linking phase in order to link program fragments. The programs are shown lying outside the file

system or database in this example; they may of course be held inside but this does not affect the method of data access.

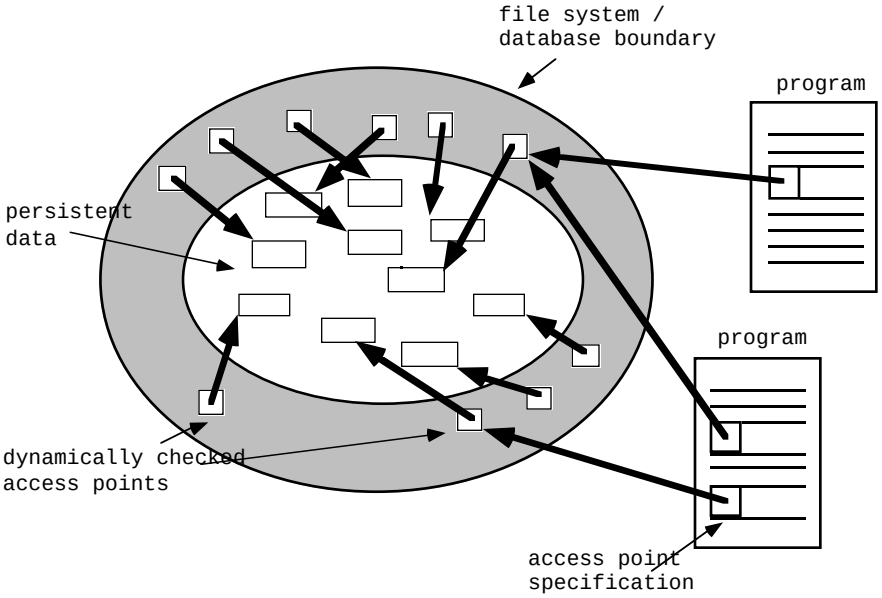

Fig. 1. Traditional access to persistent data

Figure 2 illustrates the method of using persistent data in most persistent programming systems. Programs are held outside the persistent store, commonly in a file system. The data inside the persistent store is strongly typed and interconnected by a graph of links. Now the programs contain, in addition to the names of the objects that they require, type specifications for those objects. The type specifications are represented in the figure by shaded boxes.

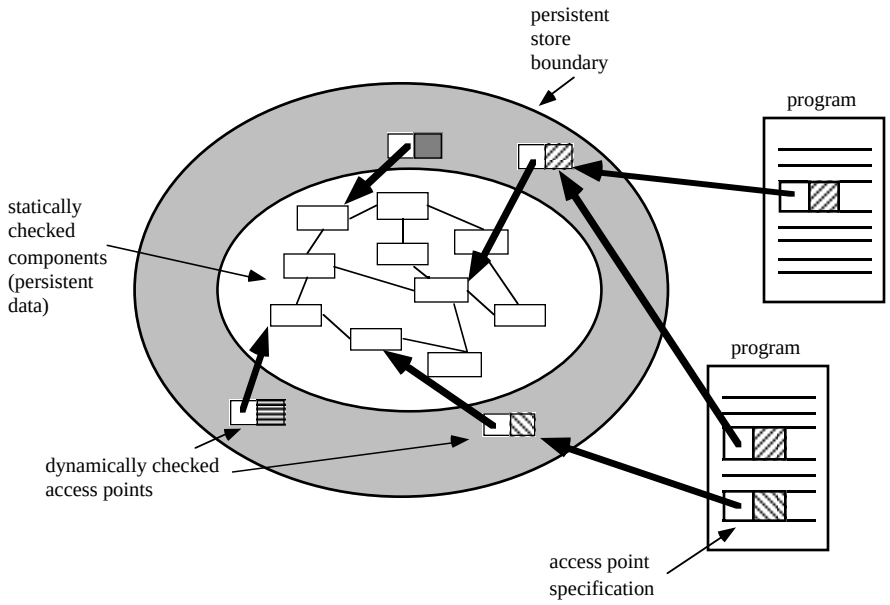

Fig. 2. Code outside the persistent store.

The programs in Figure 2 may be bound to the persistent store with largely static type checking. The graph of values inside the store may be described by purely static type definitions; the access points to this graph in the shaded area are the points of dynamic checking, about which assertions are made in programs which use the persistent data. These assertions are used by the compiler to provide static type checking for the program under the assumption that the assertions will be checked dynamically.

The collective information in the access points may be regarded as the persistent store schema. The majority of programs which use persistent data are written with respect to an unchanging partial schema description (access points which are not used by a program need not be specified). Whenever this is the case, the set of all required dynamic bindings may be organised as a prelude to the program's execution. If the execution of the binding prelude succeeds, then the execution from that point on cannot fail with a dynamic type error, and the application it represents can therefore be regarded as statically typed.

It may however be impossible to execute such programs if interim changes have been made to the structure of the persistent store and thereby the schema. Such changes may cause the execution of the binding prelude to fail since some of its assertions about the schema are no longer valid.

2.4 Hyper-programming

In contrast to these programming styles, source programs may be considered as persistent objects. In this case some of the objects accessed by a program may already be available at the time when the program is composed, so that persistent links to the objects can be included in the program instead of textual descriptions which evaluate to the links. Figure 3 shows such an environment with persistent links from the source code to the persistent data including other program fragments. Notice that some dynamic bindings may remain for flexibility but that unnecessary ones are replaced by links. By analogy with hyper-text, this style of program containing both text and links to objects is called a *hyper-program*.

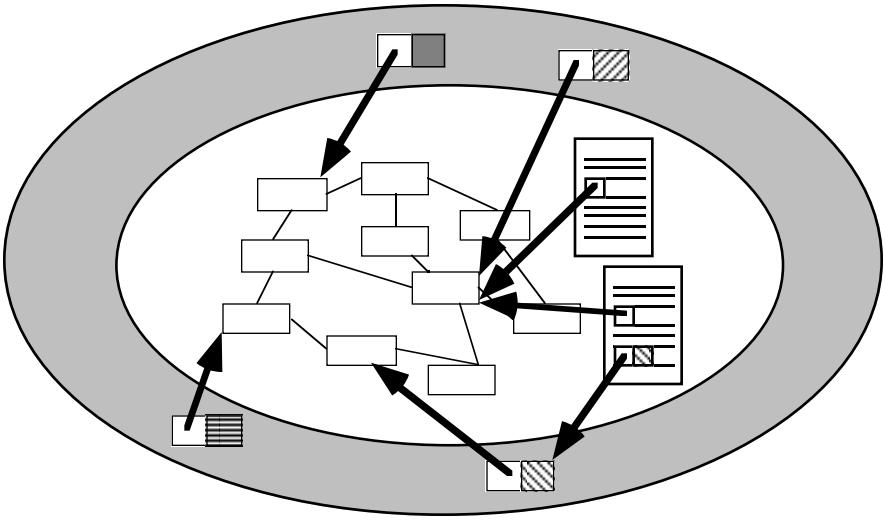

Fig. 3. Hyper-program bindings

Figure 4 shows an example of a hyper-program. The links embedded in it are represented by non-textual tokens to allow them to be distinguished from the surrounding text. The first link is to a first class procedure value *writeString* which writes a prompt to the user. The program then calls another procedure *readString* to read in a name, and then finds an address corresponding to that name. This is done by calling a procedure *lookup* to look up the address in a table data structure linked into the hyper-program. The address is then written out. Note that code objects (*readString*, *writeString* and *lookup*) are denoted using exactly the same mechanism as data objects (the table). Note also that the object names used in this description have been associated with the objects for clarity only, and are not part of the semantics of the hyper-program.

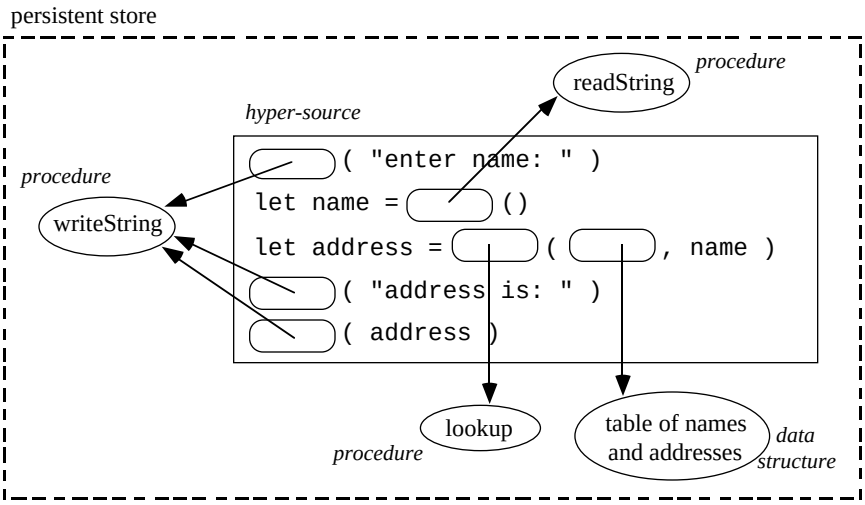

Fig. 4. A hyper-program

Figure 5 shows an example of the user interface which might be presented to the programmer by a hyper-program editing tool. The editor contains embedded light-buttons representing the hyper-program links; when a button is pressed the corresponding object is displayed in a browser window. The browser is also used to select persistent objects for linking into hyper-programs under construction.

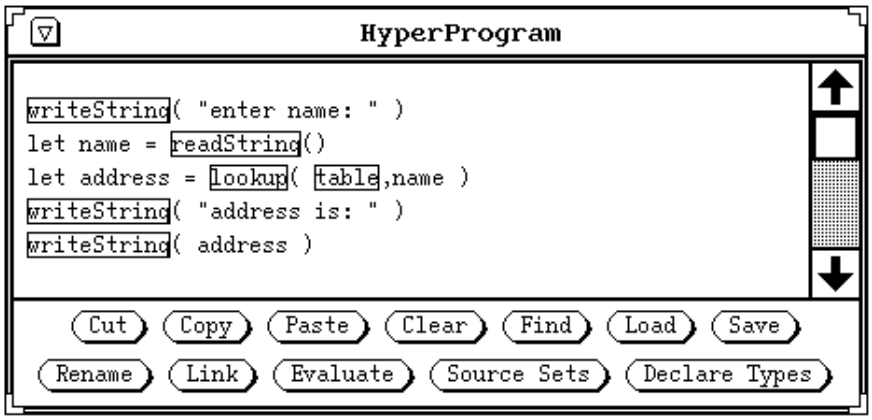

Fig. 5. User interface to a hyper-program editor

The benefits of hyper-programming are discussed in (Farkas, Dearle *et al.*, 1992; Kirby, 1992; Kirby, Connor *et al.*, 1992), and include:

- being able to perform program checking early—access path checking and type checking for linked components may be performed during program construction;

- being able to enforce associations from executable programs to source programs—links between source and compiled versions may be used;

- support for source representations of all procedure closures—free variables in closures may be represented by links, thus allowing hyper-programs to be used for both source and run-time representations of programs; and

- increased program succinctness—access path information, specifying how a component is located in the environment, and type information, may be elided.

The two extremes of programming system identified so far are file-based and hyper-programming systems. Other possibilities include a persistent system with code in the store but no persistent links from the code to other persistent objects, and a compile-time linking system in which the tokens embedded in a program are associated with data items in the persistent store when the program is compiled rather than when it is written. The linking times possible in each of these systems are shown in Table 1. From here on it will be assumed that the hyper-programming systems under consideration incorporate facilities for compile-time linking as well as composition-time linking.

Table 1. Comparison of possible linking times in various systems.

System	**Linking Time**							
	composition		compilation		linking phase		execution	
	program	data	program	data	program	data	program	data
file-based / database					•			•
persistent					•	•	•	•
compile-time linking			•	•	•	•	•	•
hyper-programming	•	•	•	•	•	•	•	•

The entries in the table may be explained as follows:

- File-based and database systems allow links to existing data to be formed only at run-time. Links to existing programs are formed during a linking phase by copying library programs into the main program. Note that this does not preserve the identity of the library programs.

- In persistent systems with first class procedures a linking phase can be implemented (Dearle, Cutts *et al.*, 1993). Since these executable programs are a form of data, linking to both programs and data can be performed either at link-time or run-time.

- Compile-time linking systems support these same linking times and also allow linking to programs and data at compilation-time. This entails the compiler accessing the persistent store to obtain the referenced data and program fragments. Compiled programs must be retained in the persistent store to maintain the referential integrity of the links (Atkinson and Morrison, 1989).

- A hyper-programming system supports all the linking times described. The programmer can specify various linking times as appropriate for different components of an application. Deciding when components should be linked into a main program involves trade-offs between program safety, flexibility and execution efficiency.

Run-time linking gives flexibility since the data (from now on, the term *data* is used to denote both programs and other kinds of data) accessed do not have to be present in the persistent store, file system or database before run-time. Indeed, the access paths to the data may not be known until run-time. Run-time program safety is low since the data may not be present when the program is run, causing a run-time failure. This kind of linking is possible in most programming systems. Execution overheads are higher than for earlier linking times, particularly in strongly typed systems where the type of the data must be checked dynamically (Connor, Brown *et al.*, 1990).

A distinct linking phase occurs between compilation and execution in many file-based systems, involving the copying of other executable or intermediate programs into the main executable program. A similar effect can also be achieved in persistent languages with higher-order procedures, where all types of data may be linked into an executable program before run-time. This provides improved safety and efficiency over run-time linking, since checks for the data's existence and type are performed before run-time. Flexibility is reduced since its use requires the data to be present earlier.

Linking at compilation-time increases safety and efficiency, bringing checks further forward in time, and reduces flexibility correspondingly. With this mechanism the data linked into an executable program is fixed during compilation.

Composition-time linking is the least flexible of the options described since the data linked to must be present at the time that the program is written. However, it offers the most safety since access to the data is always maintained once it is linked into the source code. Data access is maintained even if the source code is edited and re-compiled, which is not true of the other linking styles where editing of the source code requires all links to be re-established. Efficiency may be increased since the access path to the data need be followed only once, at composition-time, and not on every re-compilation. Programmer efficiency may also be improved since inconsistencies in the programmer's conceptual model of the persistent store may be resolved immediately while the programmer's attention is focused.

3 The Software Engineering Context

As stated earlier, the main theme of this paper is to demonstrate the advantages of persistent linking mechanisms when applied to the software engineering context. A major issue in software engineering is to keep control of all the various entities which describe a software system: that is source and executable code, along with various versions and configurations of these. In this context there are two main benefits derivable from the kind of bindings which may be used in persistent systems. Firstly, it is shown in this section how the base model of code may be simplified using hyper-programming, thereby obviating some of the requirement for the management of causations, associations, and links within a software system. It will then be outlined how activities such as version control and configuration management may be managed by tools which are themselves constructed using persistent linkage, with an associated benefit over more traditional methods.

3.1 Causations, associations and links

Several varieties of relationship between the components of a software engineering environment may be identified. These are *causations*, *associations* and *links*.

Causations are 'cause and effect' relationships. A causation between a component *A* and another component *B* is a relationship mediated by some process having *A* as input and *B* as output. This means that a change to *A* results in a corresponding but indirect change to *B*. An example of a causation is the relationship between a source program and the corresponding compiled version, mediated by a specific compiler which takes the source program as input and produces a compiled version. A modification to the source program causes a corresponding change in the compiled program but only after the process of compilation.

Fig. 6. Example of a causation

Associations are more general relationships between components. An example is an association between an executable program and the corresponding source program, maintained by a source level debugging system. This information is not intrinsic to the associated components themselves but is maintained by an external mechanism. The accuracy of associations depends on adherence to conventions: if changes to the components are made outside the control of the external mechanism, the associations may become invalid. In Figure 7, the source program could be updated without notifying the debugging system, in which case its association with the executable program would become invalid.

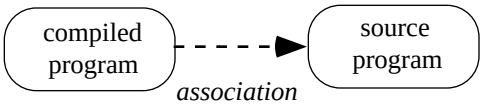

Fig. 7. Example of an association

A link, as introduced earlier, from a component *A* to another component *B* exists if a change to *B* can be immediately detected from *A* without the need for any intermediate process. In systems that support referential integrity a link from *A* to *B* always remains valid regardless of the operations performed on *B*. Links may, of course, be to L-values where the value can change without the link being altered.

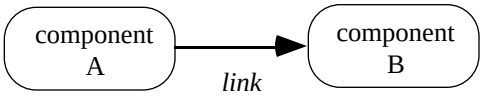

Fig. 8. Example of a link

Software environments are made safer but less flexible whenever an association can be replaced by a link. The methodology that will be described later is to replace the associations found in traditional software systems by persistent links and reverse links. This means that the environment ensures that a given software component and the objects to which it links remain accessible from one another, as a consequence of the maintenance of referential integrity.

3.2 Languages with external storage systems

Languages such as Pascal (Wirth, 1971), Ada (ANSI, 1983) and C (Kernighan and Ritchie, 1978) do not provide orthogonal persistence. Long-term data is

manipulated differently from transient data, and strong typing and referential integrity are not enforced. The data is held in a storage system, separate from the run-time environment, with which programs communicate through an interface. Examples include the Unix file system and the INGRES database system (Stonebraker, Wong *et al.*, 1976).

The program entities—such as source programs, intermediate programs and executable programs—all reside in the external storage system. Source programs are compiled to produce intermediate programs. Where necessary a linker is then used to link in existing intermediate and executable programs from a program library. This linking involves combining the intermediate program with copies of the library programs to produce a new executable program. At run-time the resulting executable program is itself copied into the data space of a run-time environment and evaluated in that context. The running program may create new data items (values and locations) with persistent links between them. It may also access existing data in the external storage system and copy data from the run-time environment to the external storage system. The run-time environment disappears at the end of execution, along with any new data items created in it.

The relationships are illustrated in Figure 9. Here solid rectangles represent source programs, rounded rectangles represent intermediate programs, diamonds represent executable programs and ellipses represent data items that can be denoted in the programming language.

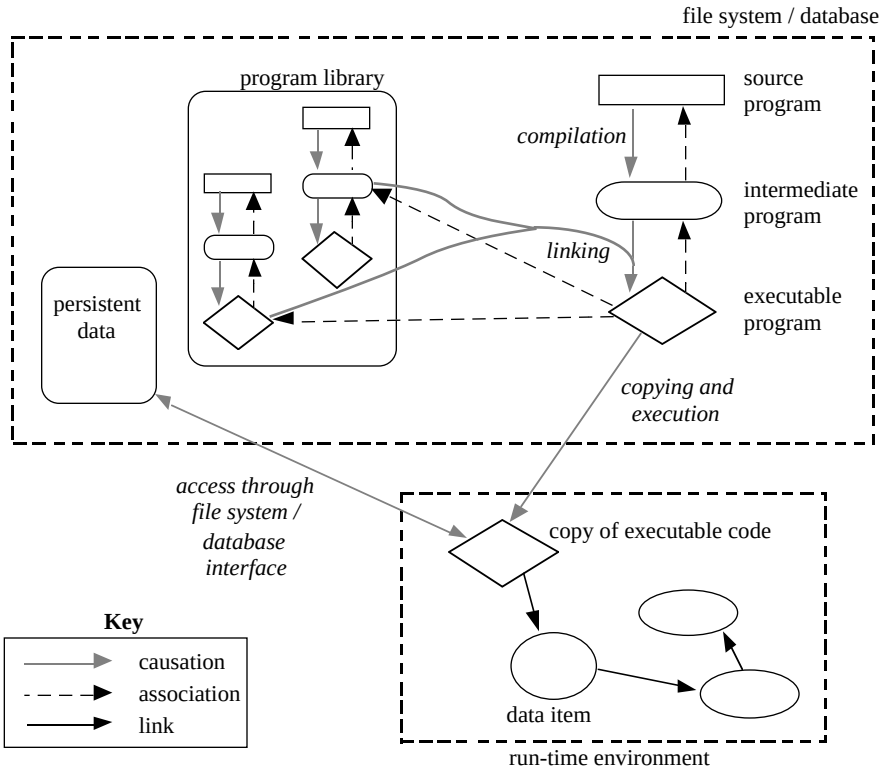

Fig. 9. Relationships in a file-based system

3.3 Hyper-programming environments

The concept of hyper-programming introduced in section 2.4 is relevant to software engineering environments in two ways. Firstly, an environment may provide enhanced facilities to support software engineering using hyper-programs rather than the traditional textual programs, as described in (Kirby, Brown *et al.*, 1994). Secondly, hyper-program technology may assist in the implementation of the environment itself.

Each hyper-program may contain links, perhaps hidden to the user, to any compiled or executable form. The compiled or executable forms may also 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 ensures that update-in-place is controlled. Such a mechanism may utilise or be enforced by the referential integrity of the persistent system. Figure 10 illustrates the relationships among software components in a hyper-programming system. Notice that the associations found in a traditional system, as illustrated in Figure 9, have been replaced by persistent links.

In the example the software component v_1 has been created by the execution of e_1. The component v_2 has a persistent link to it from both the source hyper-program and the executable form of the program. Notice also that the executable form has a reverse link to the hyper-program, and that all the programs, data and library components are persistent. The library components need not be copied or linked, simply referred to directly. Indeed they may themselves refer to other persistent values, as is shown in the persistent links (from the program library source and executable values) to v_2.

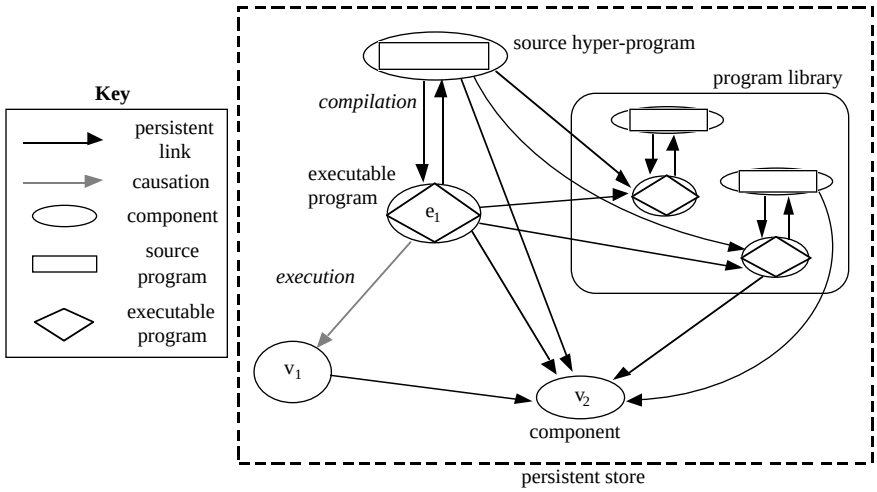

Fig. 10. Relationships in a hyper-programming system

Since the system exhibits referential integrity, updating an attribute of one of the components does not invalidate the links of the others. Changes to either the hyper-program or the executable form require new versions to be constructed, leaving the old versions intact. The hidden links from source to executable in a new version are invalid until established by compilation. Thus a new hyper-program will not be linked to by the old executable program and new executables will not be linked to the old hyper-program. Changes to hyper-programs and executables can therefore only be made in lock-step, at some suitable level of granularity (Connor, Cutts *et al.*, 1994).

The use of non-textual program representations raises a number of issues regarding the publishing of programs and transferring them to contexts outside the programming environment in which they were constructed. How, for example, to publish a program listing in a journal article? If a hyper-program is copied to another persistent environment, are the objects linked into it also copied, or are new links established to corresponding objects in the new environment? Some approaches are discussed in (Kirby, 1992; Farkas and Dearle, 1993).

4 Exploiting Persistent Linkage In Software Engineering Environments

Previous sections introduced the concept of the hyper-program, containing links to persistent objects and made possible by the provision of strong typing and referential integrity. The use of such links to persistent objects whose referential integrity is guaranteed has a number of applications in software engineering environments. Four examples are given in this section relating to:

- simplifying the programming model;
- version control;
- configuration management; and
- documentation.

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

The hyper-programming system underlying the following examples has been implemented and is in extensive use. The hyper-code, version control, configuration management and documentation systems have been prototyped. The reader should regard all of these examples as illustrative of how the mechanisms can be improved rather than as definitional in how they should be constructed. In particular, the examples make heavy use of user interfaces for illustrative purposes. The user interfaces are again prototypes and should not be confused with the application of the persistent linkage mechanisms.

4.1 Hyper-code

One of the advantages of hyper-programming listed earlier is the ability to use the hyper-program representation for both source and run-time representations of programs. At program composition time, the user may construct a hyper-program using a tool which is a combination of an editor and a browser (the editor part was illustrated in Figure 5). The editor allows text to be entered and the browser allows the persistent store to be explored for component parts of the hyper-program. When found, the components may be linked into the hyper-program, perhaps by some user gesture such as drag-and-drop. Again this saves writing code and enhances safety by early linking. Other tools may also be used to aid the construction of hyper-programs, especially where repetitive actions are frequent (Kirby, Connor *et al.*, 1994).

At run-time the hyper-program may also be used to represent an active computation. This is possible due to the non-flat nature of the hyper-program representation. Free values, that is non-local references, in objects and procedures may be represented as persistent 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 following fragment of Napier88 code creates two procedures, *inc* and *get*, which share an integer location *i*:

```
let i := 0
let inc = proc() ; i := i + 1
let get = proc( → int ) ; i
```

The first line declares the variable *i* with an initial value of 0. The second line declares the constant *inc* which is a procedure that takes no parameters, returns no result, and the effect of which is to increment *i*. The third line declares the procedure *get* which returns the current value of *i*.

Figure 11 shows the persistent links between the procedures, their hyper-program source representations and the shared location. These links are automatically created by the hyper-programming environment during execution of the code fragment.

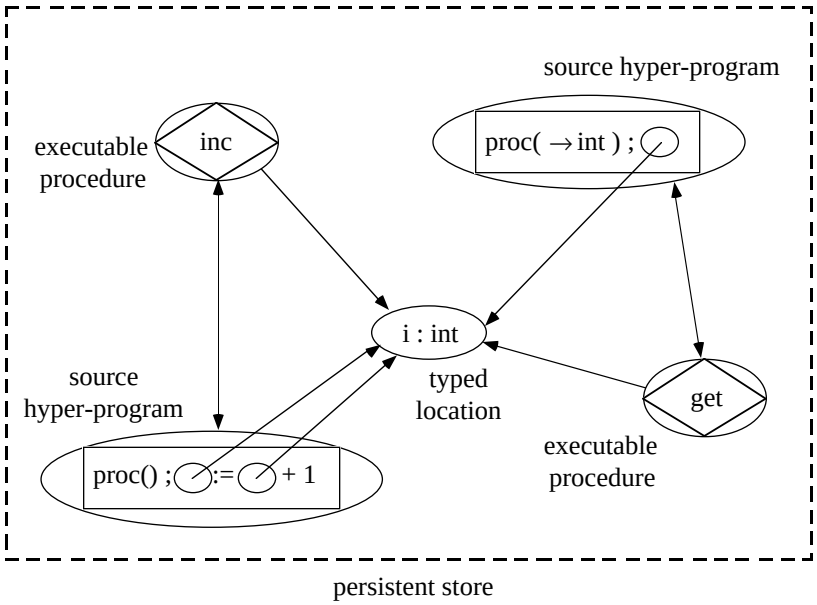

Fig. 11. Hyper-program procedure representations with a shared location

In this example the hyper-program representations are created by the underlying system. Now that the location *i* exists, it is possible for the programmer to construct a new hyper-program representation with a link to that location, and from that representation to create a new procedure value which also shares the location. In the prototype Napier88 hyper-programming system, the user interface

allows the programmer to select a graphical representation of the location *i* and then link it into a hyper-program under construction. Figure 12 shows the objects and links present after the programmer has constructed the hyper-program representation of a new procedure *dec*, and then evaluated it to create the procedure. Notice that although the representation of *inc* was created by the underlying system and that of *dec* by the programmer, they now have exactly the same form.

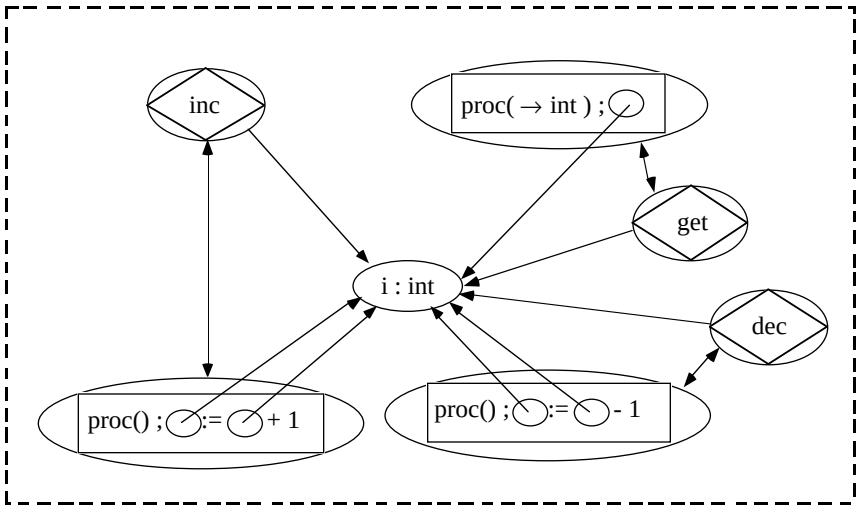

Fig. 12. New hyper-program and procedure sharing the same location

The availability of hyper-program source representations allows browsing and debugging tools to display meaningful representations of procedure closures, showing both source code and persistent 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 allows a conceptual simplification of the programming activity.

The *hyper-code* abstraction allows a single program representation form, the hyper-program, to be presented to the programmer at all stages of the software development process. In constructing a program, the programmer writes hyper-code. During execution, during debugging, when a run time error occurs or when browsing existing programs, the programmer is presented with, and only sees, the hyper-code representation. Thus the programmer need never know about those entities that the system may support for reasons of efficiency, such as object code, executable code, compilers and linkers. These are maintained and used by the underlying system but are merely artefacts of how the program is stored and executed, and as such are completely hidden from the programmer. This permits

concentration on the inherent complexity of the application rather than that of the underlying system.

In a hyper-code system it is possible to regard the source in one of two ways. In the first view of hyper-code, all values are source in the form of hyper-text and their execution appears to be that of source code interpretation. Thus the simplicity of source code interpretation may be supported by the efficiency of statically typed compiled code that is hidden from the user. In the second view of hyper-code, source code is treated not as a fundamental building block within the programming system, but instead as a transient text-based view of any value. The source does not have a conceptual permanent existence within the system, but is apparently generated from any value that may be browsed.

Figure 13 shows an example of a user interface displaying the relevant hyper-code after a run-time error or exception has occurred. The window highlights the source code expression corresponding to the point where the error occurred: in this case an attempt has been made to create a vector whose upper bound was less than its lower bound. The hyper-code displayed has exactly the same form as that which was written down when the program was constructed. If desired the programmer may press the button *vecSize* which denotes the link to a persistent store location, causing the browsing system to display the offending value. Since the hyper-code represents an existing program it is read-only; however it is possible to copy it to create a new version, or for example to write a fragment of code to assign a new value to the location *vecSize*.

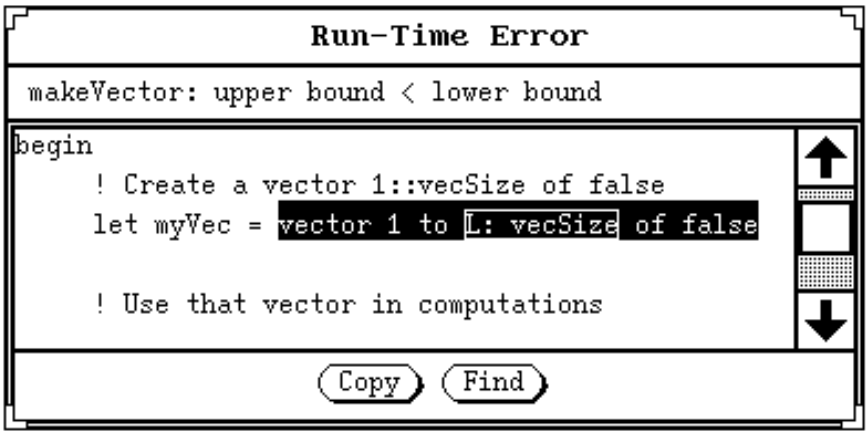

Fig. 13. Run-time errors in hyper-code

The implementation of the hyper-code model relies on the referential integrity property of the underlying persistent environment to ensure that the source hyper-program can always be obtained by traversing a persistent link from any given executable program. The hyper-code model could also be used to support source-

level debugging, with the addition of facilities for setting and removing break-points etc.
The conceptual simplicity of unified programming systems usually breaks down for one of three reasons:

- the programmer is aware of the transformations since errors are detected and reported during them;

- the programmer is aware of the transformations since they take time to perform; or

- the transformations may have to be understood in order to develop code in one context and then move it to a new context, rebinding it with the code already existing in that context.

The use of hyper-code representations therefore requires a set of tools to overcome all of these problems.

4.2 Version control

Referential integrity within a persistent system allows the traditional role of version control (Ambriola, Bendix *et al.*, 1990) to be extended to provide abstract views of the objects being versioned. One illustrative mechanism for this will now be described. It should again be emphasised that the purpose of this paper is to highlight the possibilities that arise from the use of strongly typed persistent linking with referential integrity, rather than to make a definitive statement on programming version control, configuration management and documentation. As such, the basis for any one of these systems could be one well established in the literature, with the persistent linking mechanisms used to enhance its utility.

The example 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. The decision as to what is placed under version control is left to the version builder but typically only relatively large grained application components are versioned.

To avoid confusion, the programmer responsible for maintaining the versions of a component will be referred to as the version builder, and a programmer using the software component so provided will be referred to as the version user. It should be emphasised that these roles may well overlap due to the hierarchical nature of most software systems; thus a component which is versioned may itself be constructed with the use of other versioned components. In this case the builder of one version is also the user of others.

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

Access to the privileged version builder's interface is controlled by some mechanism such as password protection (Connor, Dearle *et al.*, 1990; Morrison, Brown *et al.*, 1990) and is not discussed here.

A version controller is used to give an abstract view of a software component, providing a logical grouping of its various concrete versions. For example a version controller may group together the versions of a compiler component. 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*, which allows versions to be specified logically rather than with reference to the temporal order in which they are created (Reichenberger, 1989).

The version controller provides a number of version windows; these provide logical views of the versions and they are the only means of access to the versions for the version user. Each version window views (is mapped to) a particular version and this mapping is controlled by the version builder. Thus to access a versioned object the version 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 14 shows the structure of a particular version controller, its windows and its versions.

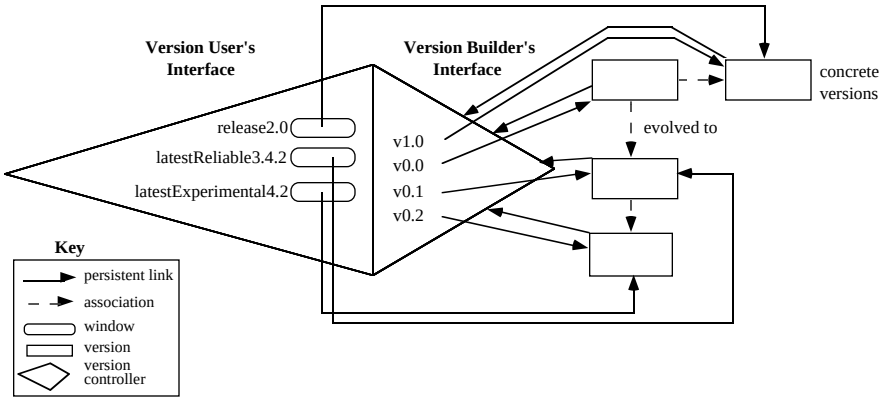

Fig. 14. A version controller

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 version builder. This is an example of a name space being layered on top of the linking graph for convenience (of the version builder).

Changing the mapping between a version window and its corresponding concrete version may only be performed by the version builder, through the protected interface. A graphical user interface could allow this to be done by gesture, by dragging a window icon from one version icon to another. For example a mapping change may be performed when a new version is created and a window corresponds to the latest version, or when some verification of a component has been carried out and a window corresponds to the latest reliable version. Other change strategies may also be used by the version builder as appropriate. Some mapping changes may be performed automatically by the version controller, for example a latest version window may be automated.

Since the version controllers may be programmed it is possible to ensure that a consistent set of changes is made over a number of version controllers. This entails one controller simulating the actions of a version builder on the other version controllers. Thus for example, changing the latest version to version 2.6 in every version controller may be performed in one atomic step. Note also that since values of any type may be versioned, a single version controller may encapsulate a large number of objects so long as they are all accessible from a single root object which is versioned.

The version controller imposes some restrictions on the way mapping changes may be performed, in order to preserve type safety. A particular version window may only ever view versions of one type, so the viewable type is fixed at the time a version window is created. This ensures type safety without the need for dynamic

type checking. If a new version has a different type from previous versions, new version windows must be created to allow access to it by the user.

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

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 (Astrahan, Blasgen *et al.*, 1976; Stonebraker, Wong *et al.*, 1976; Powell, 1985). 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.

The concurrent use of the version control system to ensure atomicity of change is orthogonal to the design of version controllers. It is provided by a transaction mechanism and is not part of the version controller.

An essential part of this technology is that versions and version windows can always identify their version controller via a persistent link, thereby replacing the associations of traditional application systems and ensuring the lock-step nature of change. This reliable link may be used later by applications and tools to identify the versions, configurations and documents from which they were constructed.

4.3 Configuration management

The presence of the persistent environment also allows re-evaluation of application configuration management. Again the mechanisms described here are illustrative of the uses of persistent linkage rather than definitive on the issue of 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 software entities, 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 again 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. There might also be a requirement for procedural specification of components.

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. Since the system has referential integrity these links can be used to discover the actual configuration of a component by inspecting the transitive closure of its hyper-code representation. Note that in the case of procedure objects, access to the hyper-code representation permits traversal of the procedure closure by traversal of a language-level data structure rather than requiring lower-level 'magic' to cross the boundary of the closure. The hyper-code provides a full representation of the closure; this implies that access to hyper-code representations by the user of the environment should be carefully controlled, since procedure closures are often used as a protection mechanism. One method of controlling this access is through a combination of existentially quantified types and password protection, as exemplified in the Napier88 standard environment (Connor, Dearle *et al.*, 1990; Kirby, Brown *et al.*, 1994).

The actual and target configurations, which may have diverged as the component evolved, can now be compared. This process is described in an example below. It is also possible to enforce target configurations on the application builders by disallowing components that do not conform to the target configuration. This may be appropriate in some circumstances but in general it is too restrictive.

The first step in constructing an application is to specify the target configuration, which may be constructed textually or by interactive gesture using a graphical representation. 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 15 shows the target configuration for a simplified compiler. Diamonds represent version controllers and rectangles represent un-versioned values. Shaded objects signify intended objects that do not yet exist or have not yet been linked to, and unshaded objects are links to existing objects. The arrows represent *intended* links only: they do not represent any actual links between components. As the target configuration develops, it is versioned and some of the shaded (intended) objects may become unshaded (existing). The intended links, however, remain since the actual relationships between the components may not be the same as the intended ones. In this example 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.

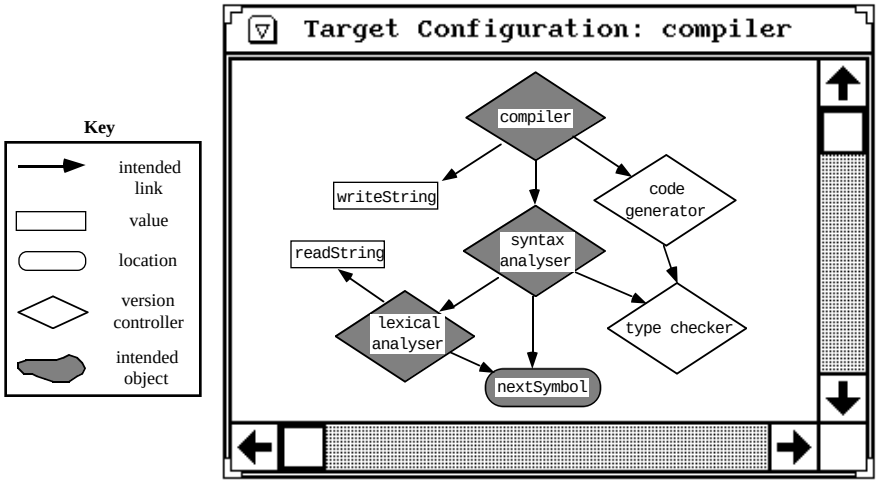

Fig. 15. Compiler target configuration

As stated earlier, the target configuration itself may be placed under version control to cater for refinements to the design. When the compiler component is created it will contain a persistent 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 16 illustrates the hyper-code for the compiler.

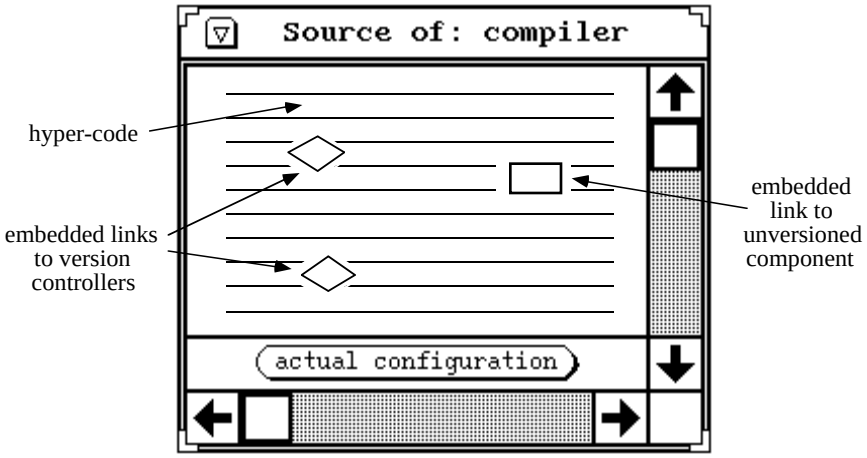

Fig. 16. The compiler source

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 17 illustrates the stage of development where the type checker and lexical analyser have not yet been constructed. Where the target configuration is used to constrain the application construction it may be used to ensure that components are only constructed in a manner consistent with the target configuration. This may be accomplished by top-down construction, but using persistent linkage it may also be enforced during bottom-up construction since the actual configuration may be calculated and compared to the target.

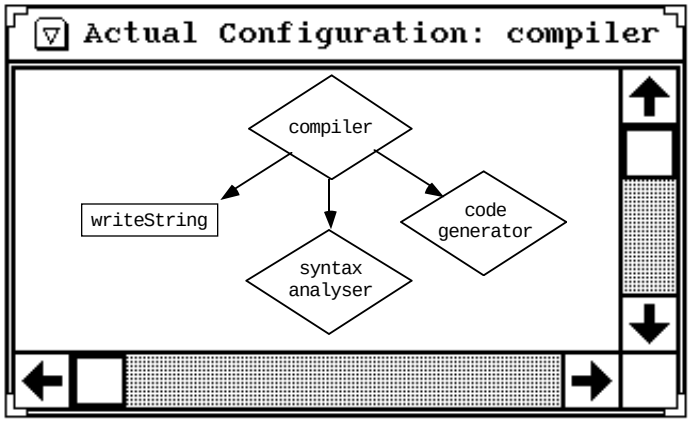

Fig. 17. The compiler actual configuration

The novelty of the actual and target configuration approaches is that by using the persistent links, real rather than intended 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. The authors know of no other system that allows an application to discover how it was configured.

4.4 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 persistent links and the relationship between application and documentation enforced by referential integrity.

Documents may contain links to objects such as 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 18 illustrates such a scheme.

The persistent 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.

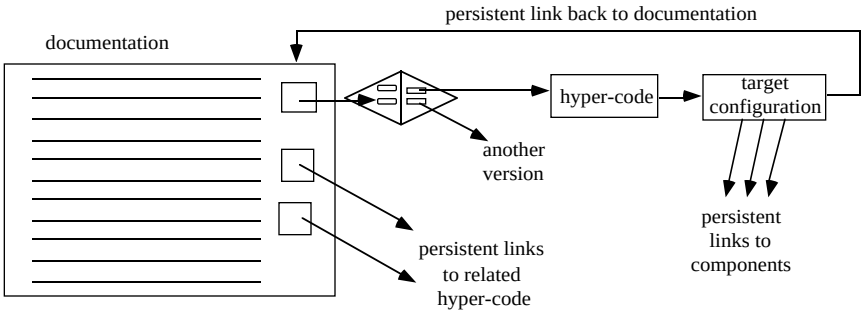

Fig. 18. System documentation with links

5 Related Work

Some of the concepts outlined in this paper have appeared in different forms in other systems. The Flex system (Currie, 1985; Stanley, 1986; Stanley and Drummond, 1988) consists of a programming language supported by a persistent

file store that contains structured data. Thus all the ingredients required for the exploitation of strongly typed persistent linkage with referential integrity are present, that is persistent links, capabilities in this case, higher order functions and strong typing since the language is a semantically complete version of algol-68. Indeed Flex also has the concept of hyper-links, called cartouches after their user interface representation. Source code may contain cartouches which point to persistent objects, in this case typed files. Furthermore the notion of pairing source with executable so that they can only be updated in lock-step is also present in Flex. The Flex system only runs on a specially microprogrammed Perq computer which perhaps explains why the concepts from it have never been published nor put into widespread use. Indeed the present authors only stumbled over this important work during the reviewing of a research grant application.

The Vesta configuration management system (Chiu and Levin, 1993; Levin and McJones, 1993) is based on a file system that does not allow overwriting in place. Since the system is Unix based, this means that there is no write permission on any file and that inodes are unique. By this means, configurations are guaranteed to refer to the same file every time since the file system ensures the integrity of the inode. To support this, the implementors re-wrote the Unix file system to remove update-in-place and thereby support configuration management by persistent links with referential integrity.

6 Conclusions

The provision of orthogonal persistence in a programming language simplifies the programming task by abstracting over the lifetime and physical location of data. Most persistent language implementations support the concept of persistence within a standard operating system environment, by adding commands to compile, link and execute programs which are represented as files within that system; executable programs then operate within the closed environment of the persistent store. This paper describes how three facilities of persistent environments, persistent linkage, strong typing and referential integrity, may be used in the construction of software environments. It is clear that the same activities can be supported by other techniques and indeed other integrated programming environments exist (Teitelbaum and Reps, 1981; Reiss, 1984; Teitelman and Masinter, 1984; Sweet, 1985; Habermann and Notkin, 1986; Dowson, 1987; O'Brien, Halbert *et al.*, 1987; Akima and Ooi, 1989; Bott, 1989; Thomas, 1989). The stronger hypothesis is that the same activities may be modelled with advantage to both the user and the system constructor in a strongly typed environment with referential integrity.

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 generally enforce referential integrity. The advantages to software environments 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 them to be presented as 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 documentation control strategy outlined is possible only because reliable links and reverse links can be inserted between documents and the objects which they describe. For simplicity, the relationships that have been described are one-to-one. The technique does however extend to many-to-many relationships.

The combination of these concepts yields a programming 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.

The presentation of this work has deliberately not touched on a number of aspects that are essential to a complete software environment, such as scale, support for requirements and design, and support for programmer teams. Rather the message has been illustrative of what can be achieved by using persistent linkage. The concept of having actual and target configurations can be extended to requirements and design. For example a requirements document may refer to other existing objects and other requirements documents and as such may become hyper-text mixed with hyper-code. The same is obviously true for design documents. However the utility of the persistent linkage can only be evaluated after experimentation.

Support for programmer teams and collaborative working requires concurrency control mechanisms that may be non-serialisable in nature. Much preliminary work is being undertaken in this area but at present solutions tend to be one-off rather than general. The Communicating Actions Control System (CACS) (Stemple and Morrison, 1992) addresses this problem within the context of persistent systems. The essence of the system is: understandability; the separation of concurrency control from data; formal capture; and a path to implementation. Global cohesion of an action in CACS is visualised as the movement of data among access sets. The visibility of the data to other users then becomes an issue of synchronisation. Where the visibility coincides with commit time, atomic transactions may be obtained, and where the visibility is immediate then synchronisation is being used. By viewing the co-ordination of the use of data as the co-ordination of the movement of the data among access sets, then the same data may be used in a different manner at different times. Thus the data may take part in an atomic transaction (Moss, 1985) one day, a saga (Garcia-Molina and

Salem, 1987) the next and may be in a co-operative computation (Nodine and Zdonik, 1992) the next. This work is at an early stage but already utilises hyper-links to ensure that specifications and implementations of the concurrency control are kept in lock step.

Scaling the notion of hyper-links is difficult in engineering terms and not always desirable in conceptual terms. Supporting hyper-links over a distributed system means that the user may be unaware of the topology of that system and that failure semantics differ from that of a local system. An obvious application of this is the use of hyper-links to support the World Wide Web. The WWW links are written in HTML and are unreliable whereas the persistent links require reliability. Mixing the descriptive technology of HTML with the reliability of persistent links provides an interesting research area. The work on structured access to files by Abiteboul, Cluet & Milo (Abiteboul, Cluet *et al.*, 1993; Abiteboul, Cluet *et al.*, 1995) provides a possible avenue for solving this problem.

The present use of persistence for constructing programming environments and software engineering tools is in its infancy but some notable experiments are being undertaken. The Glasgow Workbench controls the use of a set of software engineering tools in order to guide the programmer through the construction of persistent application systems. Present tools include application construction via design in a conventional data model (Cooper and Qin, 1992), a methodology tool, SPASM (Sjøberg, Cutts *et al.*, 1994), which tracks the dependencies between software components accurately and a thesaurus tool (Sjøberg, 1993) which aids the programmer by recording the use of names within the software system. All of these tools would benefit from the use of hyper-links to increase their safe use.

At present the hyper-programming system has been constructed, released to other users and is in regular use. The version control, configuration management and documentation systems have been prototyped and used internally. It should, however, be emphasised again that the techniques described in this paper are not meant to be definitional on how to construct version control, configuration management and documentation systems. Rather the paper illustrates how strongly typed persistent linkage with referential integrity may be used to provide new techniques for their construction and use.

Acknowledgements

We thank Malcolm Atkinson, David Stemple, John Hurst and the anonymous referees for their constructive comments regarding this paper. This work was supported by ESPRIT III Basic Research Action 6309 — FIDE$_2$, and will continue to be supported by EPSRC Grant GR/J67611. Richard Connor is supported by EPSRC Advanced Fellowship B/94/AF/1921. The original hyper-programming research was carried out in conjunction with Alan Dearle and Alex Farkas of the University of Adelaide, now at the University of Stirling. Craig Baker worked on initial prototyping of the version control system. We would also like to thank Nic Peeling, Paul Hammond and Ian Currie of the Defence Research

Agency in Malvern, England for revealing to us the secrets of the Flex programming system.

References

Abiteboul, S., Cluet, S. and Milo, T. (1993) Querying and Updating the File. In *Proc. 19th International Conference on Very Large Data Bases*, pp. 73-84, Dublin, Ireland.

Abiteboul, S., Cluet, S. and Milo, T. (1995) A Database Interface for Files Update. To appear: *Proc. ACM SIGMOD International Conference on Management of Data, San Jose, California*.

Akima, N. and Ooi, F. (1989) Industrializing Software Development: a Japanese Approach. *IEEE Software*, **6,** 13-21.

Albano, A., Bergamini, R., Ghelli, G. and Orsini, R. (1993) An Object Data Model with Roles. In *Proc. 19th International Conference on Very Large Data Bases*, pp. 39-51, Dublin, Ireland.

Albano, A., Cardelli, L. and Orsini, R. (1985) Galileo: a Strongly Typed, Interactive Conceptual Language. *ACM Trans. Database Syst.*, **10,** 230-260.

Ambriola, V., Bendix, L. and Ciancarini, P. (1990) The Evolution of Configuration Management and Version Control. *Software Eng. J.*, **5,** 303-310.

ANSI (1983) *Reference Manual for the Ada Programming Language*. Technical Report ANSI/MIL-STD-1815A, U.S. Department of Defense.

Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D. *et al.* (1976) System R: Relational Approach to Database Management. *ACM Trans. Database Syst.*, **1,** 97-137.

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

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R. (1983) An Approach to Persistent Programming. *Comp. J.*, **26,** 360-365.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R. (1984) Progress with Persistent Programming. In Stocker, P.M., Atkinson, M.P. and Gray, P.M. (eds), *Database, Role and Structure*, Cambridge University Press, Cambridge.

Atkinson, M.P. and Buneman, O.P. (1987) Types and Persistence in Database Programming Languages. *ACM Comput. Surv.*, **19,** 105-190.

Atkinson, M.P., Buneman, O.P. and Morrison, R. (1988) Binding and Type Checking in Database Programming Languages. *Comp. J.*, **31,** 99-109.

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

Atkinson, M.P. and Morrison, R. (1985) Procedures as Persistent Data Objects. *ACM Trans. Programming Languages Syst.*, **7,** 539-559.

Atkinson, M.P. and Morrison, R. (1988) Types, Bindings and Parameters in a Persistent Environment. In Atkinson, M.P., Buneman, O.P. and Morrison, R. (eds), *Data Types and Persistence*, Proc. 1st International Workshop on Persistent Object Systems, Appin, Scotland, pp. 3-20, Springer-Verlag.

Atkinson, M.P. and Morrison, R. (1989) Polymorphic Names and Iterations. In Bancilhon, F. and Buneman, O.P. (eds), *Advances in Database Programming Languages*, Proc. 1st International Workshop on Database Programming Languages, Roscoff, France (September 1987), pp. 241-246, ACM Press.

Atkinson, M.P., Morrison, R. and Pratten, G.D. (1986) A Persistent Information Space Architecture. In *Proc. 9th Australian Computing Science Conference*, Australia.

Bancilhon, F., Barbedette, G., Benzaken, V. *et al.* (1988) The Design and Implementation of O_2, an Object-Oriented Database System. In Dittrich, K.R. (ed), *Lecture Notes in Computer Science 334*, pp. 1-22, Springer-Verlag.

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

Bretl, B., Otis, A., Penney, J. *et al.* (1989) The GemStone Data Management System. In Kim, W. and Lochovsky, F. (eds), *Object-Oriented Concepts, Applications, and Databases*, Morgan-Kaufman.

Brown, A.L. (1989) *Persistent Object Stores*. Ph.D. thesis, University of St Andrews.

Bruynooghe, R.F., Parker, J.M. and Rowles, J.S. (1991) PSS: A System for Process Enactment. In *Proc. 1st International Conference on the Software Process: Manufacturing Complex Systems*.

Cardelli, L. (1985) *Amber*. Technical Report AT7T, AT&T Bell Labs, Murray Hill.

Chiu, S.-Y. and Levin, R. (1993) *The Vesta Repository: A File System Extension for Software Development*. Technical Report 106, DEC Systems Research Center.

Conklin, J. (1987) Hypertext: A Survey and Introduction. *IEEE Computer*, **20,** 17-41.

Connor, R.C.H. (1990) *Types and Polymorphism in Persistent Programming Systems*. Ph.D. thesis, University of St Andrews.

Connor, R.C.H., Atkinson, M.P., Berman, S., Cutts, Q.I., Kirby, G.N.C. and Morrison, R. (1993) The Joy of Sets. In Beeri, C., Ohori, A. and Shasha, D.E. (eds), *Database Programming Languages*, Proc. 4th International Conference on Database Programming Languages, New York City, pp. 417-433, Springer-Verlag.

Connor, R.C.H., Brown, A.B., Cutts, Q.I., Dearle, A., Morrison, R. and Rosenberg, J. (1990) Type Equivalence Checking in Persistent Object Systems. In Dearle, A., Shaw, G.M. and Zdonik, S.B. (eds), *Implementing Persistent Object Bases*, Proc. 4th International Workshop on Persistent Object Systems, Martha's Vineyard, USA, pp. 151-164, Morgan Kaufmann.

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Moore, V.S. and Morrison, R. (1994) Unifying Interaction with Persistent Data and Program. In *Proc. 2nd International Workshop on User Interfaces to Databases*, pp. 185-200, Ambleside, Cumbria.

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. and Morrison, R. (1995) Extending the Limits of Static Typing in Persistent Programming Systems. To appear: *Proc. 18th Australian Computer Science Conference, Adelaide, Australia*.

Connor, R.C.H., Dearle, A., Morrison, R. and Brown, A.L. (1990) Existentially Quantified Types as a Database Viewing Mechanism. In Bancilhon, F., Thanos, C. and Tsichritzis, D. (eds), *Lecture Notes in Computer Science 416*, Proc. 2nd International Conference on Extending Database Technology, Venice, Italy, pp. 301-315, Springer-Verlag.

Cooper, R.L. (1990a) Configurable Data Modelling Systems. In *Proc. 9th International Conference on the Entity Relationship Approach*, pp. 35-52, Lausanne, Switzerland.

Cooper, R.L. (1990b) *On The Utilisation of Persistent Programming Environments*. Ph.D. thesis, University of Glasgow.

Cooper, R.L. and Qin, Z. (1992) A Graphical Data Modelling Program With Constraint Specification and Management. In *Proc. 10th British National Conference on Databases*, Aberdeen.

Currie, I.F. (1985) Filestore and Modes in Flex. In *Proc. 1st International Workshop on Persistent Object Systems*, pp. 325-334, Appin, Scotland.

Curtis, B., Kellner, M.I. and Over, J. (1992) Process Modeling. *Commun. ACM*, **35,** 75-90.

Cutts, Q.I. (1992) *Delivering the Benefits of Persistence to System Construction and Execution*. Ph.D. thesis, University of St Andrews.

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

Dearle, A. (1988) *On the Construction of Persistent Programming Environments*. Ph.D. thesis, University of St Andrews.

Dearle, A., Cutts, Q.I. and Connor, R.C.H. (1993) Using Persistence to Support Incremental System Construction. *Journal of Microprocessors and Microprogramming*, **17,** 161-171.

Dowson, M. (1987) Integrated Project Support with Istar. *IEEE Software*, **4,** 6-15.

Farkas, A. and Dearle, A. (1993) Octopus: A Reflective Language Mechanism for Object Manipulation. In Beeri, C., Ohori, A. and Shasha, D.E. (eds), *Database Programming Languages*, Proc. 4th International Conference on Database Programming Languages, New York City, pp. 50-64, Springer-Verlag.

Farkas, A.M., Dearle, A., Kirby, G.N.C., Cutts, Q.I., Morrison, R. and Connor, R.C.H. (1992) Persistent Program Construction through Browsing and User Gesture with some Typing. In Albano, A. and Morrison, R. (eds), *Persistent Object Systems*, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy, pp. 376-393, Springer-Verlag.

Feldman, S.I. (1979) Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience*, **9,** 255-265.

Garcia-Molina, H. and Salem, K. (1987) Sagas. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 249-259.

Habermann, A.N. and Notkin, D. (1986) Gandalf: Software Development Environments. *IEEE Trans. Software Eng.*, **12,** 1117-1127.

Han, J. and Welsh, J. (1993) *Methodology Modelling: Combining Software Processes with Software Products*. Technical Report 93-17, University of Queensland Software Validation Research Centre.

Kernighan, B.W. and Ritchie, D.M. (1978) *The C programming language*. Prentice-Hall, New Jersey.

Kirby, G.N.C. (1992) *Reflection and Hyper-Programming in Persistent Programming Systems*. Ph.D. thesis, University of St Andrews.

Kirby, G.N.C., Brown, A.L., Connor, R.C.H. *et al.* (1994) *The Napier88 Standard Library Reference Manual Version 2.2*. Technical Report CS/94/7, University of St Andrews.

Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. and Morrison, R. (1992) Persistent Hyper-Programs. In Albano, A. and Morrison,

R. (eds), *Persistent Object Systems*, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy, pp. 86-106, Springer-Verlag.

Kirby, G.N.C., Connor, R.C.H. and Morrison, R. (1994) START: A Linguistic Reflection Tool Using Hyper-Program Technology. In *Proc. 6th International Workshop on Persistent Object Systems*, pp. 346-365, Tarascon, France.

Levin, R. and McJones, P.R. (1993) *The Vesta Approach to Precise Configuration of Large Software Systems*. Technical Report 105, DEC Systems Research Center.

Matthes, F. and Schmidt, J.W. (1992) *Definition of the Tycoon Language TL - A Preliminary Report*. Technical Report FBI-HH-B-160/92, University of Hamburg, Germany.

Morrison, R., Bailey, P.J., Brown, A.L., Dearle, A. and Atkinson, M.P. (1985) The Persistent Store as an Enabling Technology for an Integrated Project Support Environment. In *Proc. 8th IEEE International Conference on Software Engineering*, pp. 166-172, London.

Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. and Atkinson, M.P. (1987) Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment. *Software Eng. J.*, December, 199-204.

Morrison, R., Brown, A.L., Connor, R.C.H. *et al.* (1994) *The Napier88 Reference Manual (Release 2.0)*. Technical Report CS/94/8, University of St Andrews.

Morrison, R., Brown, A.L., Connor, R.C.H. *et al.* (1990) Protection in Persistent Object Systems. In Rosenberg, J. and Keedy, J.L. (eds), *Security and Persistence*, Proc. International Workshop on Security and Persistence, Bremen, 1990, pp. 48-66, Springer-Verlag.

Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P. (1988) Flexible Incremental Binding in a Persistent Object Store. *ACM SIGPLAN Notices*, **23,** 27-34.

Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P. (1990) On the Classification of Binding Mechanisms. *IP Letters*, **34,** 51-55.

Moss, J.E.B. (1985) *Nested Transaction: An Approach to Distributed Computing.* MIT Press, Cambridge, Massachusetts.

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

Nodine, M.H. and Zdonik, S.B. (1992) Co-operative Transaction Hierarchies: Transaction Support for Design Applications. *VLDB Journal*, **1,** 41-80.

O'Brien, P.D., Halbert, D.C. and Kilian, M.F. (1987) The Trellis Programming Environment. In *Proc. International Conference on Object-Oriented*

Programming Systems, Languages and Applications (OOPSLA'87), pp. 91-102, Orlando, Florida.

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

PS-algol (1988) *PS-algol Reference Manual, 4th edition*. Technical Report PPRR-12-88, Universities of Glasgow and St Andrews.

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

Reiss, S.P. (1984) Graphical Program Development with PECAN Program Development Systems. *ACM SIGPLAN Notices*, **19,** 30-41.

Rochkind, M.J. (1975) The Source Code Control System. *IEEE Trans. Software Eng.*, **SE-1,** 364-370.

Schaffert, C., Cooper, T. and Wilpot, C. (1985) *Trellis Object-Based Environment Language Reference Manual*. DEC Systems Research Center.

Sjøberg, D.I.K. (1993) *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems*. PhD thesis, University of Glasgow.

Sjøberg, D.I.K., Cutts, Q.I., Welland, R. and Atkinson, M.P. (1994) Analysing Persistent Language Applications. In *Proc. 6th International Workshop on Persistent Object Systems*, pp. 227-247, Tarascon, France.

Stanley, M. (1986) *An Evaluation of the Flex PSE*. Technical Report 86003, Defence Research Agency, Malvern, England.

Stanley, M. and Drummond, P.D. (1988) *A Flexible Basis for Software Configuration Management*. Technical Report 4127, Defence Research Agency, Malvern, England.

Stemple, D. and Morrison, R. (1992) Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach. In *Proc. 15th Australian Computer Science Conference*, pp. 873-891, Hobart, Tasmania.

Stonebraker, M., Wong, E., Kreps, P. and Held, G. (1976) The Design and Implementation of INGRES. *ACM Trans. Database Syst.*, **1,** 189-222.

Sweet, R.E. (1985) The Mesa Programming Environment. In *Proc. ACM SIGPLAN Symposium on Programming Languages and Programming Environments*, pp. 216-229.

Teitelbaum, T. and Reps, T. (1981) The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM*, **24,** 563-573.

Teitelman, W. and Masinter, L. (1984) The Interlisp Programming Environment. In Barstow, D.R., Shrobe, H.E. and Sandewall, E. (eds), *Interactive Programming Environments*, McGraw-Hill, New York.

Thatte, S.M. (1986) Persistent Memory: A Storage Architecture for Object Oriented Database Systems. In *Proc. ACM/IEEE International Workshop on Object-Oriented Database Systems*, pp. 148-159, Pacific Grove, California.

Thomas, I. (1989) Tool Integration in the Pact Environment. In *Proc. 11th International Conference on Software Engineering*, pp. 13-22.

Tichy, W.F. (1985) RCS - A System for Version Control. *Software – Practice and Experience*, **15,** 637-654.

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

Wirth, N. (1971) The Programming Language Pascal. *Acta Informatica*, **1,** 35-63.