This paper should be referenced as:

Morrison, R., Baker, C., Connor, R.C.H., Cutts, Q.I. & Kirby, G.N.C. “Approaching Integration in Software Environments”. University of St Andrews Technical Report CS/93/10 (1993).

Approaching Integration in Software Environments

R. MORRISON, C. BAKER, R.C.H. CONNOR, Q.I. CUTTS AND G.N.C. KIRBY
Department of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland

Persistent programming systems are generally recognised as the appropriate 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. An essential attribute of persistent systems is the referential integrity of data. That is, once a link (reference) to an object is established, it persists over time. As a consequence no object can be deleted while another refers to it. 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 how having 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 persistent data across the hitherto enclosed boundaries of system components.

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[4, 5]. As a result a number of persistent systems have been developed including PS-algol[1], Napier88[28], Galileo[2], TI Persistent Memory System[40], Amber[16] and Trellis/Owl[35]. 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[6-11, 14, 18, 20, 21, 26-28, 41]. 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.

The persistence abstraction has been recognised as the appropriate 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[25, 27]. Object-Oriented Database Systems such as GemStone[13] and O_2[12] have at their core a persistent object store; process modelling systems use a persistent base to preserve their modelling activities over execution sessions[15].

Recently persistent programming systems have been developed that allow the complete software process to take place entirely within the persistent environment[13, 28]. Thus each component of the software process can take advantage of the persistent environment. This paper focuses on one particular advantage, that of referential integrity. Although other integrated programming environments have been developed[30, 33, 37-39], either they do not support referential integrity or it is not used to gain the benefits described here.

The feature of referential integrity supported by the persistent environment 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, it enables naming schemes that operate by convention to be replaced by links to persistent objects. The term *link* will be used here instead of *reference* to avoid the connotation that *reference* has in other contexts. These linking possibilities extend to inter-component relationships and allow 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 naming conventions are replaced by links, the representations of the software components are non-flat. This is because the representations themselves contain direct links to other components rather

than symbolic names for them. In the context of documentation, this leads to hyper-text[17, 29], where documents contain links to other documents. In the context of programs it leads to the concept of the hyper-program[24], where program representations contain direct links to data objects of any type including other program components.

2. 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 for as long as the link exists. In a strongly typed persistent environment, such as Napier88[28], this also means that the type correctness of the links will be 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 ceases to exist only if the last link to the object is removed (cf the UNIX link command). Thus although the object may not be available for new links to be made to it, all extant links still point to it. In systems without explicit deletion, garbage collection may be used to determine when an object may be finally removed.

Using links with referential integrity may 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 it is guaranteed for as long as the link exists. For example, the configuration management tool *make*[22] and the version control tool *SCCS*[34] both rely on a name space to identify the components of an application. If the conventions of the name space are improperly used (such as changing the name of a file from outside the tool) the tool will fail. This possibility could be prevented if the tools used links instead of names.

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. Clearly 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.

The use of names and their role in the software construction process may now be re-examined.

3. CAUSATIONS, ASSOCIATIONS AND LINKS

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

*A close analogy in UNIX is referring to objects by *inode* number rather than a path name.

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 the 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.

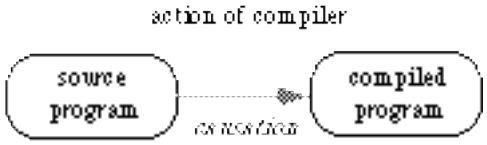

Figure 1. Example of a causation

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 the example the source program could be updated without notifying the debugging system, in which case its association with the executable program would become invalid.

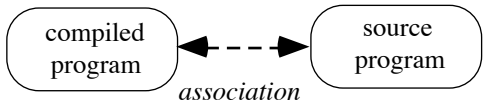

Figure 2. Example of an association

Links are references between components. A link 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*.

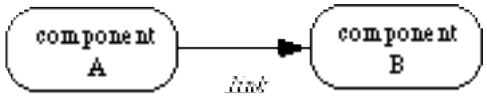

Figure 3. Example of a link

The software process is made safer whenever an association can be replaced by a link. The methodology that will be described in Section 4 is to replace the associations found in traditional software systems by links and reverse links. This means that the software components and the objects to which they link are in lock-step, guaranteed by the referential integrity of the persistent system.

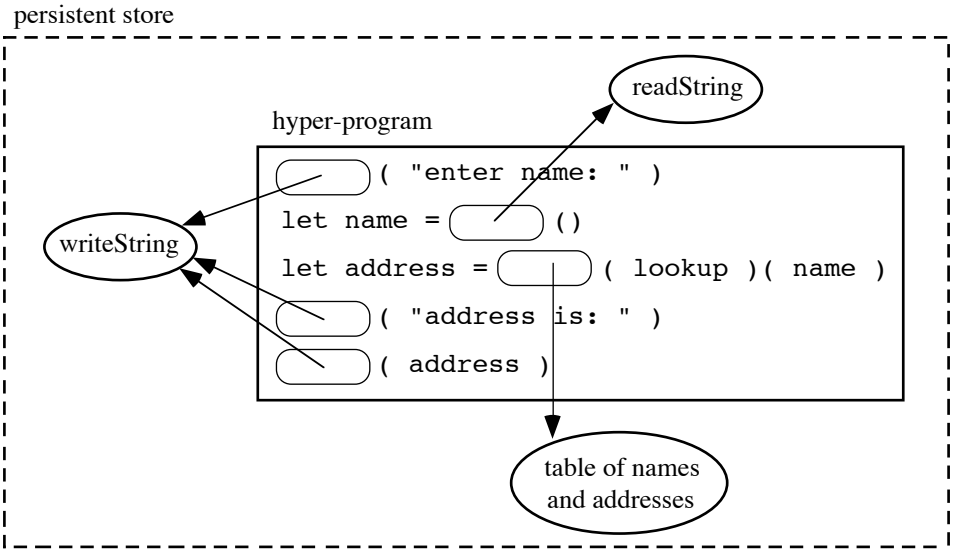

Figure 4. A hyper-program

For simplicity, most of the relationships in Section 4 are one-to-one. The technique does however extend to many-to-many relationships.

4. NEW PARADIGMS FOR THE SOFTWARE PROCESS

This section shows how the use of links to persistent objects whose referential integrity is guaranteed gives rise to new techniques for the software process. Four examples are given: 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.

4.1 Hyper-programming

Traditionally programs are represented as linear sequences of text. Where a program accesses another object during its execution, it contains a textual description of that object, describing how to locate the object. At some stage during the software process the description is resolved to establish a link to the object itself. Commonly this occurs during linking for code objects and during execution for data objects, and the environment in which the resolution takes place varies accordingly. There is no guarantee that a textual description of an object will remain valid until the time of its resolution, even if it is valid at the time that the program is written.

In such systems programs are constructed and stored in some long-term storage facility, such as a file system, separate from the run-time environment which disappears at the end of each program execution. By contrast, in persistent systems, programs may be constructed and stored in the same environment as that in which they are executed. This means that objects accessed by a program may already be available at the time that the program is composed. In this case 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 4 shows an example of a hyper-program. The links embedded in it are represented by light buttons which when pressed will cause the corresponding objects to be displayed. 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).

More than one program may contain links to a particular component, and the graph of program components can become highly interconnected. The benefits of hyper-programming are discussed in[23]. They include:

- being able to perform program checking early—

  access path checking and type checking for linked components may be performed at program construction;

- support for source representations of all procedure closures—

  free variables in closures may be represented as links, thus hyper-programs may be used for both source and run-time representations of programs;

- being able to enforce associations from executable programs to source programs—

  links between source and compiled versions may be used;

- increased program succinctness—

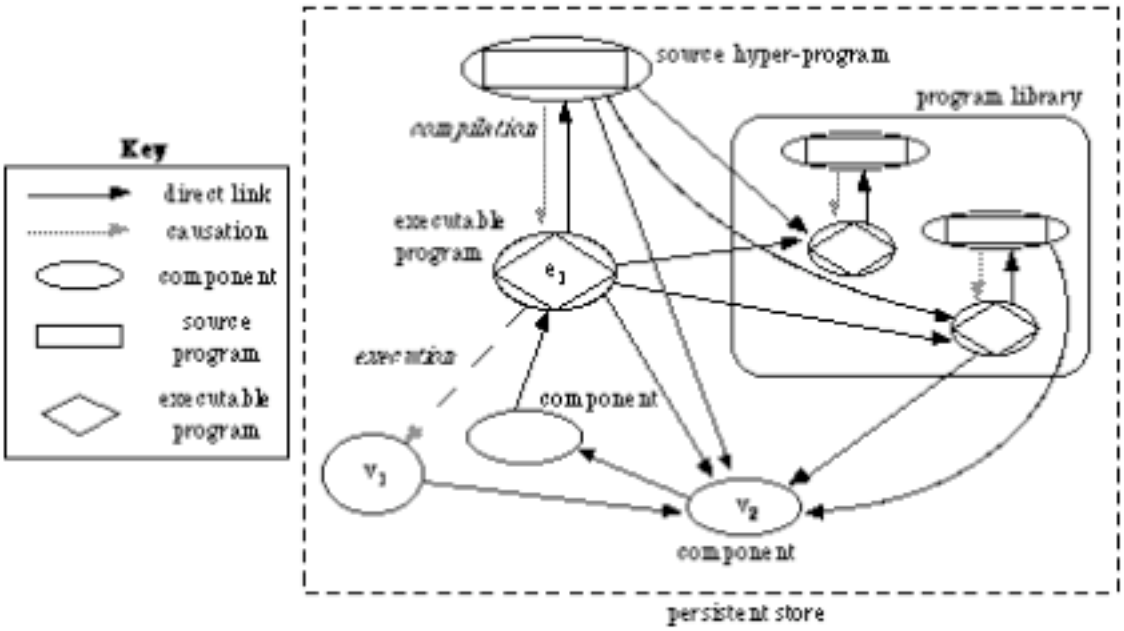

Figure 5. Relationships in a hyper-programming system

access path information, specifying how a component is located in the environment, may be elided; and

- increased ease of program composition—

  links may be inserted by programmer gesture as well as by typing.

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. Update in place is controlled by editing a copy of a component's source code and replacing both the component and its source code in one atomic step. Figure 5 illustrates the relationships among software components in a hyper-programming system. Notice that the associations found in a traditional system have been replaced by links.

In Figure 5, the software component v_1 has been created by the execution of e_1. The component v_2 has a 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.

Since the system exhibits referential integrity, altering 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, perhaps with the same identity, to be constructed. 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.

4.2 Hyper-code

One of the advantages of hyper-programming listed above 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 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 included in the hyper-program by some user gesture, such as drag and drop. Again this saves writing code, enhances safety by early linking and promotes software re-use.

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 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

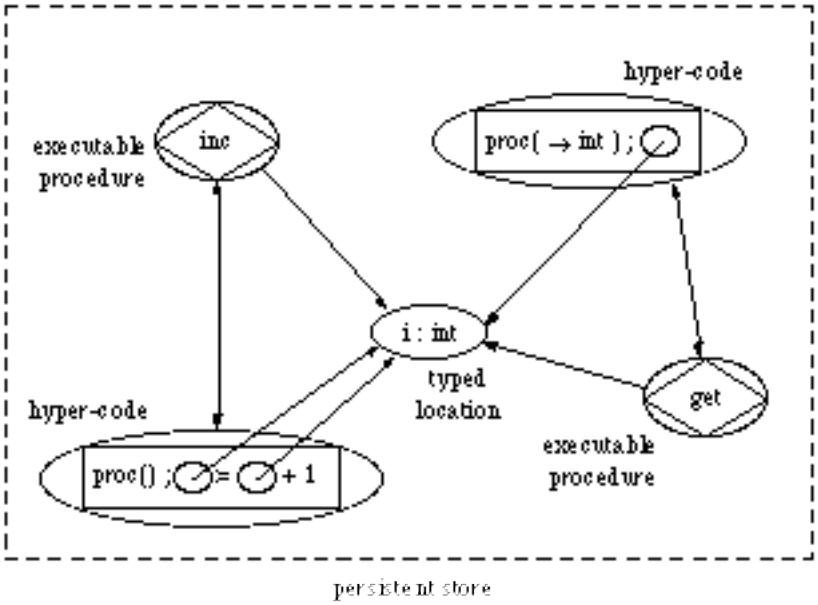

Figure 6. Hyper-code procedure representations with a shared location

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. These are maintained and used by the underlying system but are 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.

Figure 6 shows the links between two procedures, their hyper-code source representations and a shared location. Notice that the hyper-code could be a representation constructed by the programmer, if location *i* already existed, or by program execution, or a combination of both. The distinction is immaterial.

Programmers may copy hyper-code, compose hyper-code using text and links, install hyper-code in the persistent store and run hyper-code. The mechanical aspects, such as compilation and linking, are maintained by the system thereby relieving the programmer or configurer of the burden of maintaining associations. The conceptual simplification of hyper-code is that the programmer only sees one representation of the program throughout its life cycle.

4.3 Version control

Different styles of version control are provided in different systems. Referential integrity within a persistent system allows the traditional role of version control to be extended to provide an abstract view of the objects being versioned. One mechanism for this 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. The decision as to what is placed under version control is left to the application builder but typically only relatively large grained application components are versioned.

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.

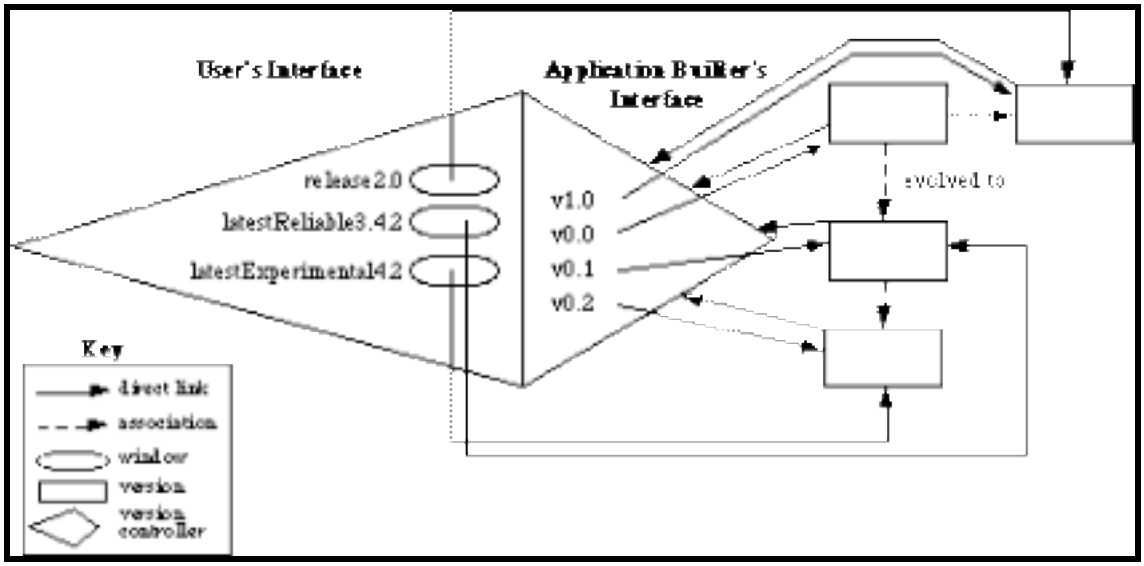

Figure 7. A version controller

Access to the privileged application builder's interface is controlled by some mechanism such as password protection[19].

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*. This allows versions to be specified logically rather than with reference to the temporal order in which they were created[32].

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 user. 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 7 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. A graphical user interface allows 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, if a window corresponds to the latest version, or when some verification of a component has been carried out, if a window corresponds to the latest reliable version. Other change strategies may also be used by the application builder as appropriate. Some mapping changes may be performed automatically by the version controller, for example a latest version window may be automated.

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 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 windows must be created to allow access to it by the user.

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.

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 [3, 31, 36]. 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 link, thereby replacing the associations of traditional systems and ensuring the lock-step nature of change.

4.4 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. 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. 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 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 8 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.

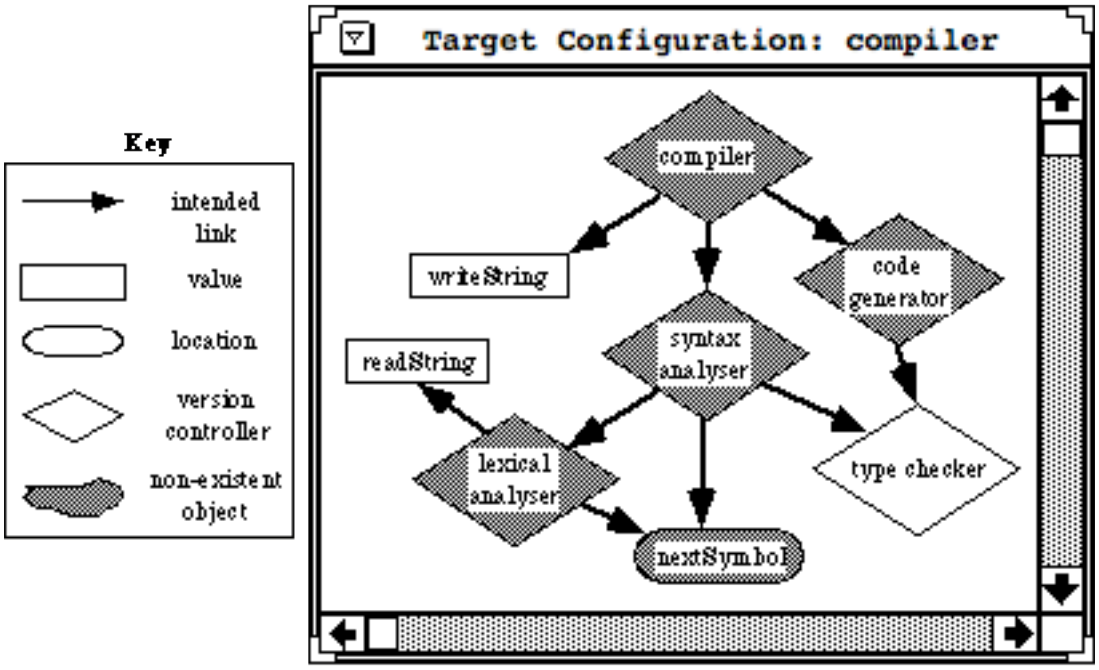

Figure 8. Compiler target configuration

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 9 illustrates the hyper-code for the compiler.

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

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

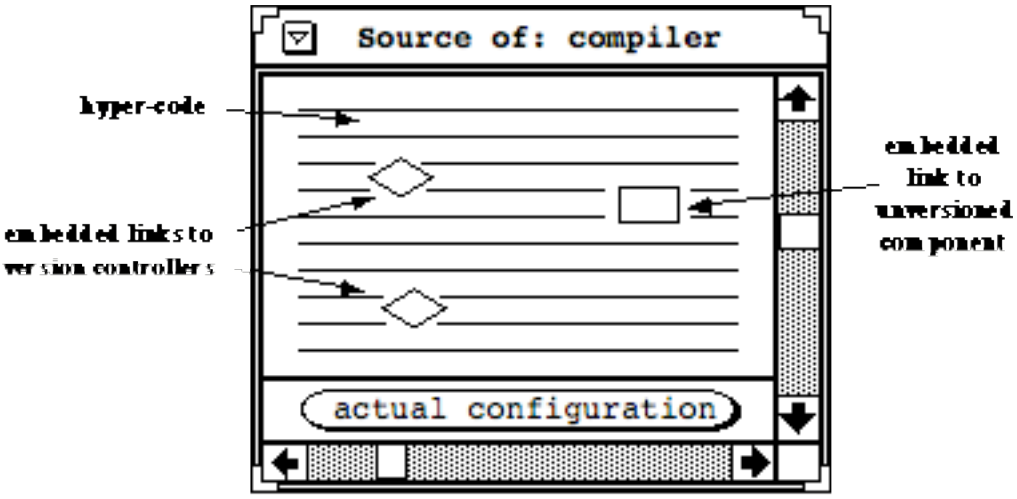

Figure 9. The compiler source

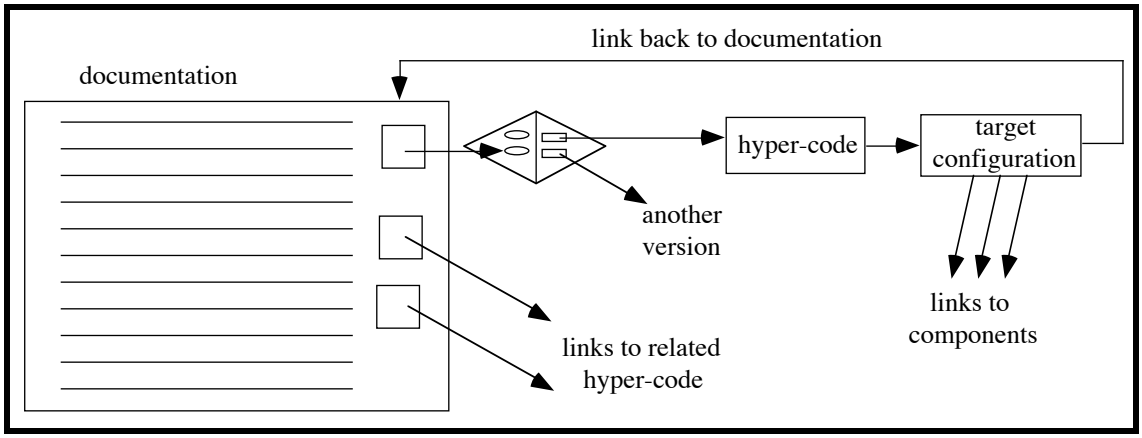

Figure 11. System documentation with links

all the system components may now be placed under version control, generic configurations may be constructed from which families may evolve.

4.5 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 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 11 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.

5. 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 an approach to the provision, within the persistent environment, for the entire software process. It is clear that the same activities can be supported in both environments. The stronger hypothesis is that the same activities may be modelled with advantage in a strongly typed environment with referential integrity.

Orthogonally persistent environments are by definition strongly typed, highly structured, and

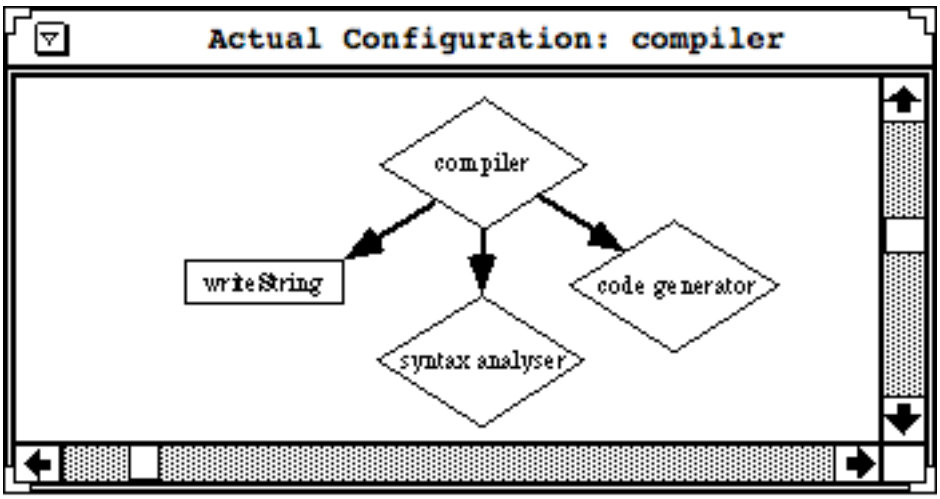

Figure 10. The compiler actual configuration

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 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 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.

Acknowledgements

We thank David Stemple for his constructive comments regarding this paper. This work was supported by ESPRIT III Basic Research Action 6309 – $FIDE_2$. The original hyper-programming research was carried out in conjunction with Alan Dearle and Alex Farkas of the University of Adelaide.

REFERENCES

1. PS-algol Reference Manual, 4th edition. Technical Report PPRR-12-88, Universities of Glasgow and St Andrews (1988).
2. A. Albano, L. Cardelli and R. Orsini, Galileo: a Strongly Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems* **10** (2), 230-260 (1985).
3. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Waid and V. Watson, System R: A Relational Approach to Database Management. *ACM Transactions on Database Systems* **1** (2), 97-137 (1976).
4. M. P. Atkinson, Programming Languages and Databases. In *Proceedings of the 4th IEEE International Conference on Very Large Databases*, pp. 408-419 (1978).
5. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott and R. Morrison, An Approach to Persistent Programming. *Computer Journal* **26** (4), 360-365 (1983).
6. M. P. Atkinson, P. J. Bailey, W. P. Cockshott, K. J. Chisholm and R. Morrison, Progress with Persistent Programming. Technical Report PPRR-8-84, Universities of Glasgow and St Andrews (1984).
7. M. P. Atkinson and O. P. Buneman, Types and Persistence in Database Programming Languages. *ACM Computing Surveys* **19** (2), 105-190 (1987).
8. M. P. Atkinson, K. J. Chisholm and W. P. Cockshott, PS-algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices* **17** (7), 24-31 (1982).
9. M. P. Atkinson and R. Morrison, Procedures as Persistent Data Objects. *ACM Transactions on Programming Languages and Systems* **7** (4), 539-559 (1985).
10. M. P. Atkinson and R. Morrison, Types, Bindings and Parameters in a Persistent Environment. In *Data Types and Persistence* (eds. M. P. Atkinson, O. P. Buneman and R. Morrison), pp. 3-20, Springer-Verlag, (1988).
11. M. P. Atkinson, R. Morrison and G. D. Pratten, A Persistent Information Space Architecture. In *Proceedings of the 9th Australian Computing Science Conference*, Australia (1986).
12. F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard and F. Valez, The Design and Implementation of O2, an Object-Oriented Database System. In *Lecture Notes in Computer Science 334* (ed. K. R. Dittrich), pp. 1-22, Springer-Verlag, (1988).
13. B. Bretl, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, M. Williams and D. Maier, The GemStone Data Management System. In *Object-Oriented Concepts, Applications, and Databases* (eds. W. Kim and F. Lochovsky), Morgan-Kaufman, (1989).
14. A. L. Brown, Persistent Object Stores. *Ph.D. Thesis,* University of St Andrews (1989).
15. R. F. Bruynooghe, J. M. Parker and J. S. Rowles, PSS: A System for Process Enactment. In *Proceedings of the 1st International Conference on the Software Process: Manufacturing Complex Systems* (1991).
16. L. Cardelli, Amber. Technical Report AT7T, AT&T Bell Labs, Murray Hill (1985).
17. J. Conklin, Hypertext: A Survey and Introduction. *IEEE Computer* **20** (9), 17-41 (1987).
18. R. C. H. Connor, Types and Polymorphism in Persistent Programming Systems. *Ph.D. Thesis,* University of St Andrews (1990).

19. R. C. H. Connor, A. Dearle, R. Morrison and A. L. Brown, Existentially Quantified Types as a Database Viewing Mechanism. In *Lecture Notes in Computer Science 416* (eds. F. Bancilhon, C. Thanos and D. Tsichritzis), pp. 301-315, Springer-Verlag, (1990).
20. A. Dearle, Constructing Compilers in a Persistent Environment. In *Proceedings of the 2nd International Workshop on Persistent Object Systems*, Appin, Scotland (1987).
21. A. Dearle, On the Construction of Persistent Programming Environments. *Ph.D. Thesis,* University of St Andrews (1988).
22. S. I. Feldman, Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience* **9** 255-265 (1979).
23. G. N. C. Kirby, Reflection and Hyper-Programming in Persistent Programming Systems. *Ph.D. Thesis,* University of St Andrews (1992).
24. G. N. C. Kirby, R. C. H. Connor, Q. I. Cutts, A. Dearle, A. M. Farkas and R. Morrison, Persistent Hyper-Programs. In *Persistent Object Systems* (eds. A. Albano and R. Morrison), pp. 86-106, Springer-Verlag, (1992).
25. R. Morrison, P. J. Bailey, A. L. Brown, A. Dearle and M. P. Atkinson, The Persistent Store as an Enabling Technology for an Integrated Project Support Environment. In *Proceedings of the 8th IEEE International Conference on Software Engineering*, pp. 166-172, London (1985).
26. R. Morrison, A. L. Brown, R. C. H. Connor, Q. I. Cutts, G. N. C. Kirby, A. Dearle, J. Rosenberg and D. Stemple, Protection in Persistent Object Systems. In *Security and Persistence* (eds. J. Rosenberg and J. L. Keedy), pp. 48-66, Springer-Verlag, (1990).
27. R. Morrison, A. L. Brown, R. C. H. Connor and A. Dearle, Polymorphism, Persistence and Software Reuse in a Strongly Typed Object-Oriented Environment. *Software Engineering Journal* (December), 199-204 (1987).
28. R. Morrison, A. L. Brown, R. C. H. Connor and A. Dearle, The Napier88 Reference Manual. Technical Report PPRR-77-89, University of St Andrews (1989).
29. J. Nielsen, Hypertext and Hypermedia. Academic Press, New York (1990).
30. P. D. O'Brien, D. C. Halbert and M. F. Kilian, The Trellis Programming Environment. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pp. 91-102, Orlando, Florida (1987).
31. M. S. Powell, Adding Programming Facilities to an Abstract Data Store. In *Proceedings of the 1st International Workshop on Persistent Object Systems*, pp. 139-160, Appin, Scotland (1985).
32. C. Reichenberger, Orthogonal Version Management. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pp. 137-140, Princeton, New Jersey (1989).
33. S. P. Reiss, Graphical Program Development with PECAN Program Development Systems. *ACM SIGPLAN Notices* **19** (5), 30-41 (1984).
34. M. J. Rochkind, The Source Code Control System. *IEEE Transactions on Software Engineering* **SE-1** (4), 364-370 (1975).
35. C. Schaffert, T. Cooper and C. Wilpot, Trellis Object-Based Environment Language Reference Manual. Technical Report 372, DEC Systems Research Center (1985).
36. M. Stonebraker, E. Wong, P. Kreps and G. Held, The Design and Implementation of INGRES. *ACM Transactions on Database Systems* **1** (3), 189-222 (1976).
37. R. E. Sweet, The Mesa Programming Environment. In *Proceedings of the ACM SIGPLAN Symposium on Programming Languages and Programming Environments*, pp. 216-229 (1985).
38. T. Teitelbaum and T. Reps, The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Communications of the Association for Computing Machinery* **24** (9), 563-573 (1981).
39. W. Teitelman and L. Masinter, The Interlisp Programming Environment. In *Interactive Programming Environments* (eds. D. R. Barstow, H. E. Shrobe and E. Sandewall), McGraw-Hill, New York (1984).
40. S. M. Thatte, Persistent Memory: A Storage Architecture for Object Oriented Database Systems. In *Proceedings of the ACM/IEEE International Workshop on Object-Oriented Database Systems*, pp. 148-159, Pacific Grove, California (1986).
41. F. Wai, Distribution and Persistence. In *Proceedings of the 2nd International Workshop on Persistent Object Systems*, pp. 207-225, Appin, Scotland (1987).