This paper should be referenced as:

Kirby, G.N.C. & Morrison, R. “Orthogonal Persistence as an Implementation Platform for Software Development Environments”. University of St Andrews Technical Report CS/97/6 (1997).

Orthogonal Persistence as an Implementation Platform for Software Development Environments

G. N. C. Kirby & R. Morrison

Department of Mathematical and Computational Sciences, University of St Andrews,
St Andrews, Fife KY16 9SS, Scotland.

{graham, ron}@dcs.st-and.ac.uk

Abstract

Software development environments need to maintain representations of software entities and the relationships between them. Various supporting software platforms have been used to provide the storage for these entities and relationships, including file systems and databases. This paper describes how a persistent object system may be used as such an implementation platform. It shares many of the advantages of object oriented databases, with the additional ability to model first-class code values, and to introduce new type descriptions dynamically.

1 Introduction

One of the main functions of a software development environment (SDE) is to store and organise information about various aspects of software production. This includes entities such as program source code, compiled code, executable code, documentation, specifications, configurations, information about authorised users and so on. Each of these may be versioned. In addition, the environment must keep track of the relationships between the entities. For example, for a given source program it may be required to be able to determine which compiled code entities are derived from it, which are the relevant documentation entities, what is its version history etc. The E-R diagram in Figure 1 shows some of the information which might be stored.

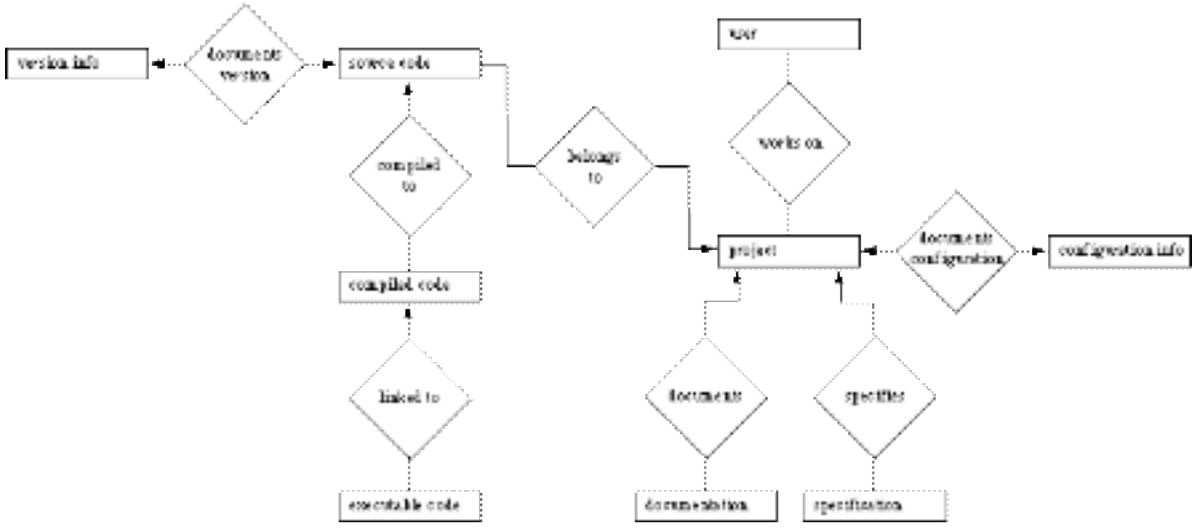

Figure 1. E-R Diagram for a Hypothetical Software Development Environment

The significant point here is not the details of the particular entities and relationships, but rather that there are a number of relationships which must be maintained by the software development environment in order for it to operate correctly. From a given entity it should always be possible to locate the other entities to which it is related.

Ensuring that such relationships are always represented correctly involves several aspects. The environment must ensure, firstly, that the entities themselves are not altered in an inconsistent manner, and secondly, that the mechanism for retrieving the related entities from a given entity is not corrupted. The degree to which such constraints can be enforced by the environment depends to some degree on the nature of the repository used to store the representations of the entities and relationships. Where the repository itself provides protection mechanisms it may be relatively easy for the environment above it to make guarantees. Where little protection is provided at the lower levels it requires greater implementation effort for the SDE to enforce the necessary constraints, and in some cases it may be necessary for users to conform to certain conventions in order for the SDE to operate correctly.

No SDE can give 100% reliability since, in the extreme case, the machine or machines on which it is stored may all fail simultaneously. Choosing a particular implementation platform for an environment is thus a question of degree: to provide a given level of robustness will require a different amount of implementation effort on different platforms. Of course other factors will also be affected, such as performance and flexibility.

The aim of this paper is to point out the suitability of a persistent object system (POS) as an implementation platform for SDEs, to compare the support it provides with that of alternative platforms, and to mention some interesting research directions which are made possible.

2 Related Work

A number of SDEs and software development tools [Roc75, Fel79, Tic85] are based around file systems. Thus the representations of the entities are stored in files, and the associations between entities are either implicit in the names of the files and their positions in the directory hierarchy, or are stored explicitly in auxiliary files. File-specific information maintained by the file system, such as creation and modification times, and access control flags, are used by the environments. Such environments have the advantage that the many existing file-based tools can continue to be used.

Such systems often rely on file naming conventions. For example a Pascal source file named *prog.c* may not be processed correctly. They may also be vulnerable to misuse if the files are manipulated by a user directly through the file system interface rather than through the environment. For example a source file might be deleted, leaving a compiled program with no corresponding source, or the source file might be replaced by another with the same name but containing a completely different program. Accidental misuse of this nature can largely be avoided if users are sufficiently disciplined to use only the environment's interface; problems may arise however when that is used in combination with the lower level file system interface.

Other drawbacks of using an existing file system are that it may be difficult to implement adequate concurrency control mechanisms above the file system layer, and the difficulties of storing structured entities within files: the environment implementation has to flatten the structure to a byte sequence for writing to a file, and rebuild the structure when it is read in.

The Cedar environment [SZH85] is implemented above a custom-built file system layer which is only accessed through the Cedar language. This allows the problems described above to be avoided but involves a large implementation effort.

The Vesta system [LM93] provides version control and configuration management facilities based on a conventional file system. It avoids the problems of misuse by implementing a higher user layer which gives the abstraction of immutable files. From the user view, all versioning proceeds by copying and no files are ever updated. The implementers state that one of the main implementation difficulties was ensuring atomic update to a group of files [CL93] .

Most newly developed SDEs use databases rather than file systems to store entity data, with the same motivations as for introducing databases to other applications: these include improving resilience, security, performance and concurrency control. Systems have been built above relational databases [CC83, PL83] ; these do indeed provide these advantages, but are still not well suited to storing highly structured data as may be necessary for entities such as configuration and version information, for example.

Increasingly common now is the use of object oriented databases (OODBs) to provide the storage layer. These have the advantage of being suited to the storage of highly structured entity representations, and have been used successfully in such systems as Arcadia [Kad92] and Forest [JV95] . OODBs by their nature provide type system protection which may be used to restrict inappropriate actions on entity representations. Most also allow the integrity of links between objects to be enforced, which assists in the reliable implementation of relationships between entities. The importance of typing and safe links at the SDE support layer was also recognised in the ECMA PCTE definition [LM93] .

Conceptually, the objects stored in an OODB contain both data and the code (methods) that operates on the data. In practice however, since all objects belonging to a particular class share the same methods, the method code is associated with the class definition rather then the objects . Thus methods are not first class objects and it is not possible, for example, for a method to generate and return a new method. Schema evolution mechanisms are necessary if the code is to be allowed to change dynamically.

In an orthogonally persistent programming system this distinction between code and data is removed, with code values having the same civil rights as all other data. This allows the use of some new techniques for software construction and maintenance. Specifically, *hyper-programming* involves the embedding of typed links to persistent objects within source programs, rather than using dynamically resolved textual denotations of those objects. This is described in Section 4, after a summary of orthogonal persistence in Section 3.

3 Orthogonal Persistence

Persistent programming languages were developed in an effort to reduce the burden on the application programmer of organising the transfer of long-term data between volatile program storage and non-volatile storage [ABC+83] . Previously, application data which was to be retained between activations had to be written explicitly to a database or file system, and later read in again to the application space. The flattening and rebuilding of data structures that this required involved a significant programming overhead, and an increased intellectual effort since the programmer had to keep track of a three way mapping between program representation, database/file representation and real world. The introduction of orthogonally persistent languages meant that any program data could be made persistent simply by identifying it as such, with all transfers between memory hierarchy layers handled transparently. This simplified matters for the application programmer: the three way mapping was reduced to a single mapping between program representation and real world, as illustrated in Figure 2.

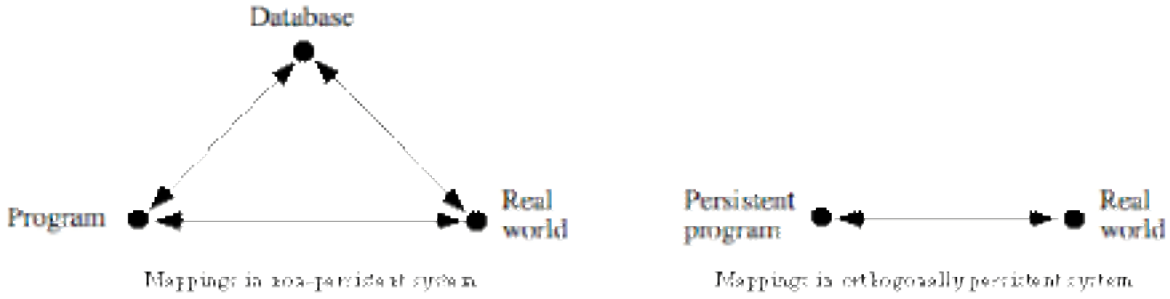

Figure 2. Conceptual simplification afforded by orthogonal persistence

Not only did the automatic transfer of data reduce work for the programmer, it also allowed more varieties of data to be kept between program activations. Because of the flattening process needed for explicit transfer of data structures to file or database representations, there were some forms of data which could not be made persistent in this way, since they could not be fully flattened by application-level programs. Examples included objects (in the object-oriented sense) and procedures with encapsulated state, abstract data types, and the identities of objects or values. Thus orthogonal persistence means that all of the features provided by a programming language can be used effectively in long-lived applications rather than only in toy systems. See [AM95] for an overview of current persistence research.

4 Persistence and Software Development Environments

The benefits of a POS as a platform for implementing software development environments derive principally from the following features: referential integrity; strong typing and first-class code values.

4.1 Referential Integrity

The property of referential integrity, provided by most orthogonally persistent systems, ensures that dangling object references can never occur. Thus once a reference to a given object is established the system guarantees that the object will remain available via that reference for as long as the reference exists. This property can be supported by combining a garbage collected volatile memory with a "persistence by reachability" implementation which automatically makes persistent all objects reachable via the transitive closure of one or more persistent roots. Objects cannot be explicitly deleted by the programmer but are removed from the system once they can no longer be accessed.

Referential integrity is a useful property for constructing reliable applications in general, since a possible failure mode is removed. In the context of supporting software development environments it greatly simplifies the problem of maintaining the various relationships illustrated in Figure 1, since all that is required is for references to related entities to be stored in the representation of each entity. The POS then ensures that the related entities will be stored for as long as the entities are accessible.

It is impossible for a user to use a lower abstraction layer to accidentally corrupt a relationship stored in this way, since the POS provides no such interface. Malicious corruption is possible but would require low level access to disk storage, and being able to interpret low level storage formats. In this respect POSs provide similar support to that of OODBs, in contrast to file systems which provide little support for referential integrity.

4.2 Strong Typing

Most POSs support rigorously enforced and relatively sophisticated type rules. A strongly typed system is one in which all code is checked for type correctness before it is allowed to execute. The checking may be static, if it is possible to determine correctness simply by analysis of the program text, or dynamic if it is necessary to access the run-time environment in order to verify the correctness of an operation. Typically both static and dynamic checking are required: as much as possible is checked statically, in order to detect errors early and to improve efficiency by factoring out run-time checks, while the facility for dynamic checking where necessary is retained for flexibility [Con90] .

The enforcement of a type system by a SDE platform enables the SDE implementer to define types for the various entities supported, and if the type system is sufficiently sophisticated (supporting an object model, modules, first class procedures or similar) these can capture the operations allowed on each kind of entity. Thus the implementation platform ensures that the user can only invoke the legal operations defined on each entity.

POSs and OODBs allow type descriptions of arbitrary complexity to be associated reliably with entity representations. In contrast, file system platforms usually support only the encoding of limited type information in file name suffixes, and even these may be contravened by a user changing a file name through the file system interface. File system tools often attempt to deduce further type information by examination of file contents but this may be unreliable since files are not always self-describing. For example it may be impossible to determine whether a file contains a C or C++ source program from its contents.

4.3 Introducing New Code

Most POSs support first-class code values in the form of procedures. Thus procedures may be nested, passed as parameters, returned as procedure results, stored in data structures and made persistent in the same manner as other values. This provides a flexible programming model which may be used to support, among others, information hiding, views, protection and separate compilation [AM84] .

In the context of supporting SDEs, the ability to represent executable code within the platform's type system helps in modelling as many aspects as possible of the software process. Consider how the platform captures the simple action of compiling a (self-contained) source code entity to produce an executable code entity, which is then executed. This requires definitions of types for source and executable code entities. A simple string representation would suffice for the former, while the latter must represent, at the least, the types of any parameters passed to the new executable code and of any result returned, in order that the type correctness of its execution can be verified. This requires flexibility on the part of the SDE platform since the type definition for the executable code needs to be introduced dynamically, during the compilation process. The sequence of events might be as follows:

- source code entity created;
- source code successfully compiled;
- type information for executable code derived from source code;
- new type information introduced into SDE;
- executable code entity created;
- new code executed.

This is difficult to achieve with those OODB platforms which provide a static type schema against which all code must compiled. Such systems do not cater naturally for the dynamic generation and introduction of new type definitions during the execution of an application (the SDE in this context). The extension of OODBs to cope with dynamic schema evolution is an active research area [Bra93, MS93, Odb95] .

Although it is obviously simple to support such a process on a file system platform, this is at the cost of being able to associate only limited type information with the various entities, as discussed in the previous section.

A POS is well suited as a platform for this process due to its support for strong but dynamic typing and first class procedures. Dynamic typing makes it possible for the SDE to generate new type definitions, derived from compilation of source entities, and have them safely introduced into the system without disrupting existing objects. Executable code entities can then be represented as procedures, with the normal associated type information. In this scenario the SDE tools such as compilers and linkers are available as procedures in the same way as user programs. This approach is similar to that of the file-based Vesta system, where the tools used to create a configuration are themselves specified within the system.

4.4 Hyper-Programming

The treatment of source programs as strongly typed persistent objects, which is made possible by the use of a POS or OODB as the support platform, permits a new approach to program construction. Hyper-programming involves storing strongly typed references to other persistent objects within a source program representation. Thus the source code entity is represented by a graph rather than a linear text sequence. By analogy with hyper-text this is called a hyper-program [MCC+95] . It may be considered as similar to a closure, in that it contains both a textual program and an environment in which non-locally declared names may be resolved. The difference is that with a hyper-program the environment is explicitly constructed by the programmer who specifies persistent objects to be bound into the hyper-program at construction time.

Figure 3 shows an example of a hyper-program representation. The first embedded reference 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 the name. This is done by calling a procedure *lookup* to look up the address in a table package referred to by the hyper-program. The address is then written out. Note that code objects (*readString*, *writeString* and *lookup*) are treated in exactly the same way 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.

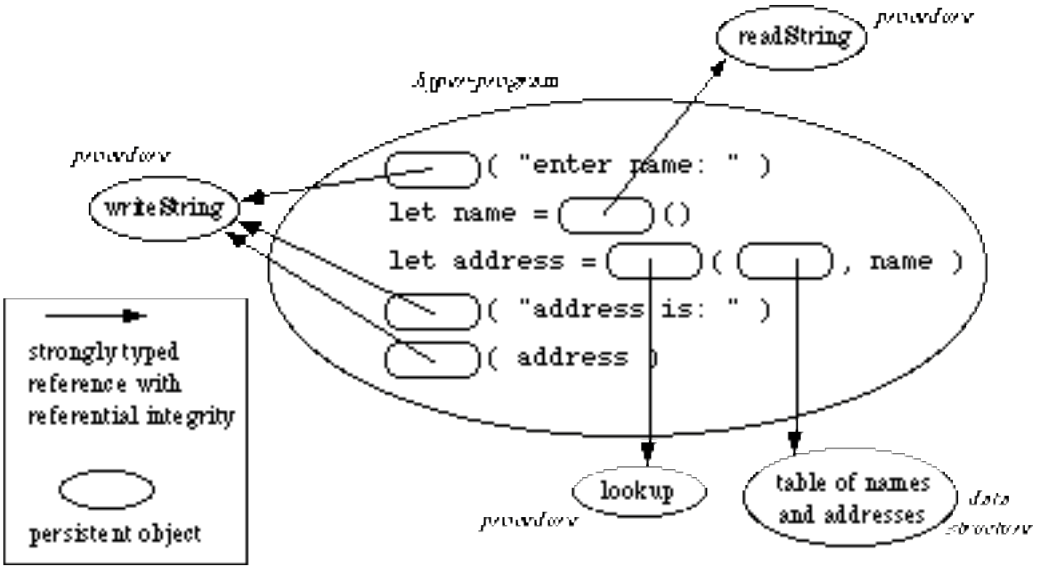

Figure 3. Hyper-program represented as persistent object graph

Figure 4 shows an example of a simple user interface to a hyper-program editor [KBC+96] . The editor displays light-buttons embedded within the text representing the hyper-program references; when a button is pressed the corresponding object is displayed in a separate graphical object browser window. The browser is also used to select persistent objects for embedding in hyper-programs under construction. A more sophisticated SDE might also allow objects to be retrieved by query over the persistent store.

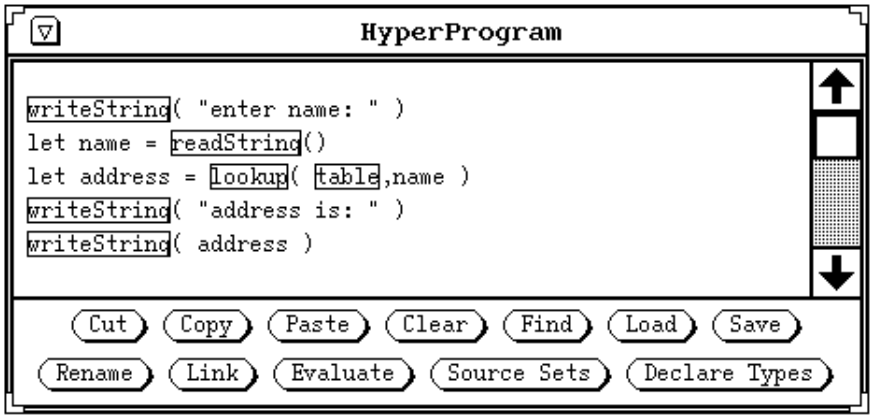

Figure 4. User interface for hyper-program editor

The support of hyper-program construction techniques by a POS provides a number of advantages to the user of an SDE based on it:

- Program succinctness: textual descriptions of the locations and types of persistent components used by the program may be replaced by simple embedded references.

- Increased execution efficiency: checking of the validity of specified access paths to other components is factored out when they are embedded directly in the source program. Checking of type consistency may be performed at compilation time rather than execution time.

- Reliable access to components: where a textual description of a component is replaced by a direct reference, the underlying referential integrity of the POS ensures that the component will always be accessible by the program. By contrast, where a textual description is used it may become invalid by the time the program executes, even if it was valid at the time the program was constructed.

- Automatic source code retention: the hyper-program notation may also be used to represent procedure closures, with encapsulated state denoted by direct references embedded in the text. This means that it is possible to associate a source representation with every procedure value, which fully captures its state. This source representation can be recorded automatically by the POS when the procedure is created, and permanently associated with the procedure by recording a reference to it in the closure value.

Due to the source retention facility the SDE is guaranteed that the hyper-program source code will always be trivially available from any procedure value. The representation of procedure state as embedded references gives a mechanism for accessing the internal state of existing applications which may be used by SDE tools such as debuggers, documentation generators etc.

The availability of first-class procedures supports an application construction style in which each application is composed of a graph of procedures which refer to each other and to application-specific data. Since the POS allows the hyper-program source code, including embedded references, to be retrieved from any particular procedure, the configuration details of an application are an intrinsic part of the application itself. As such they may be reliably accessed by the SDE.

In the example shown in Figure 5 the application is invoked via the main procedure labelled *A*, which calls procedures *B* and C. They in turn call other procedures and access data internal to the application. To discover the configuration of the application the SDE first obtains the hyper-program source code for procedure *A*. This is a data structure containing

the text and a list of embedded references, in this case to *B* and *C*. Their hyper-programs are retrieved in turn, and the full configuration is eventually derived by traversing the entire graph.

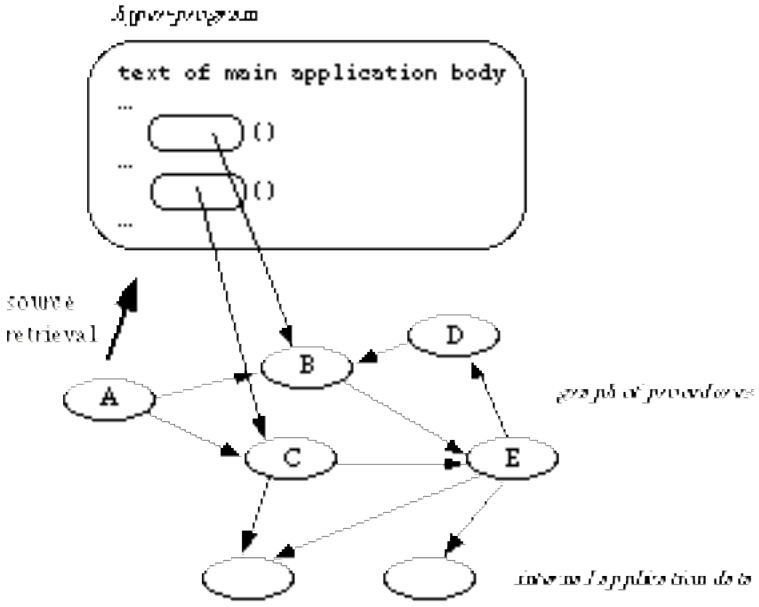

Figure 5. Application configured from graph of procedures

In this way the SDE is relieved of the task of maintaining configuration information for applications since it is performed by the POS support platform. A similar approach could be used to embed versioning information and documentation within the application itself.

Since the programmer has access to an application's internal configuration, it becomes possible to alter the implementation of a particular procedure component of the application without having to completely rebuild the application and losing the current application state. This is achieved by obtaining a hyper-program representation of the required procedure, editing the textual part, compiling to produce a new procedure which contains references to the same application state, and finally installing it in place of the existing procedure. An application architecture which supports this methodology is described in [DCC93] .

The provision of source code browsing and editing tools in the SDE is facilitated by the availability of a structured source form. For example it is relatively simple to transform the hyper-program obtained from an application into a hyper-text form suitable for browsing[1]. Of course it may not be desirable to allow all SDE users unrestricted access to hyper-program forms since the internal application state may be intentionally hidden.

All of the features described in this section could be provided by an OODB platform if, but only if, it supported the dynamic introduction of new types or classes. This is necessary to allow the compilation of hyper-programs constructed within the OODB into executable programs with statically unknown types.

---

[1]For an example see: *http://www-ppg.dcs.st-and.ac.uk/NapierSource/*

5 Further Work

A prototype SDE has been developed, based on the Napier88 persistent object system[2]. This supports the construction of hyper-programs and automatic source code retention. Currently only simple software development tools have been implemented. Tools for managing versioning, configurations and documentation will be developed in the future. The approach for these will be similar to that currently used for applications and hyper-program source code: all entities which need to be associated reliably will be linked with typed references between them.

Another research direction involves making use of the tight association between executable procedures and hyper-program source code to provide a simplified SDE interface to the programmer. The existence of those entities which are not essential to the software construction task, such as executable program forms, will be hidden from the programmer. Thus the distinction between source program and executable program will be removed: the programmer will interact with a single representation, the hyper-program, for all SDE activities, including program construction, execution, debugging, profiling etc.

We also intend to experiment with the provision of hyper-programming on a more widely used platform, possibly on a persistent version of Java such as PJava, currently under development at Glasgow University [Jor96] .

6 Conclusions

Software development environments have been implemented on a variety of software platforms, including file systems, object oriented databases and persistent object systems. These platforms vary in the amount of support provided for reliable storage of software entities, concurrency control, ensuring consistent update, storing relationships between entities and so on. Clearly the greater the support provided in the underlying platform, the less has to be implemented in the SDE, so long as that support is appropriate to the requirements of the SDE.

This paper has attempted to show how the important features of a POS—strong typing, referential integrity, first-class code and the ability to introduce new types dynamically—make it suitable as a support platform on which SDEs may be implemented. POSs differ markedly from file system platforms in their support for typed objects and referential integrity. The distinction between POSs and OODBs is less strong but the significant differences in this context are their support for first-class code values, and, in comparison with many OODBs, their greater flexibility in allowing new types to be introduced during application execution.

Thus the use of a POS as a support platform allows more of the aspects required of an SDE to be implemented in the support platform and mapped directly into the SDE, rather than needing to be implemented at the SDE level. This is illustrated in Figure 6. The maintenance of relationships between entities is shown spanning the two layers for object oriented databases, since these platforms do support referential integrity but may not be able to satisfactorily represent all entity types.

---

[2]Available at: *http://www-ppg.dcs.st-and.ac.uk/Info/Release2.2.1.html*

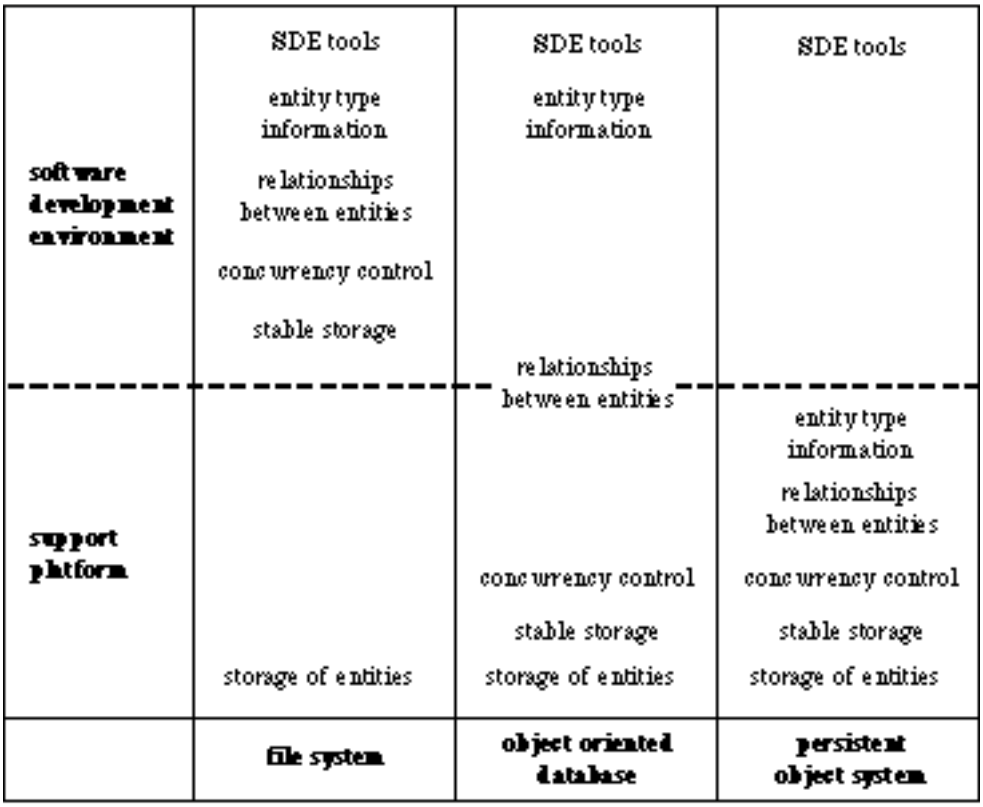

Figure 6. Comparison of SDE features supported by various platforms

We postulate that provision of such features at the support platform level rather than the SDE level will simplify the implementation of SDEs and improve SDE reliability since it reduces the scope for unwanted access to entities through lower abstraction layers. It should also allow greater flexibility for the SDE implementer since the need for entity naming conventions is removed. Finally, the potential of hyper-program technology, made possible by these features, has been outlined.

One likely disadvantage of this approach is a reduction in portability of SDE implementations, since they will make more assumptions about the support layer. This, and the claimed advantages, will need to be tested in a substantial SDE built using a POS.

7 Acknowledgements

We thank Roy Levin for his help in our understanding of the Vesta configuration management system. This work is partially supported by UK EPSRC grant GR/J 67611 "Delivering the Benefits of Persistence to System Construction". The research prototype mentioned in this paper was developed by the persistent programming group at St Andrews which included Dharini Balasubramaniam, Richard Connor, Quintin Cutts, Vivienne Dunstan, Dave Munro and Stephan Scheuerl.

8 References

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

[AM84] Atkinson, M.P. & Morrison, R. "Persistent First Class Procedures are Enough". In **Lecture Notes in Computer Science 181**, Joseph, M. & Shyamasundar, R. (ed), Springer-Verlag (1984) pp 223-240.

[AM95] Atkinson, M.P. & Morrison, R. “Orthogonally Persistent Object Systems”. VLDB Journal 4, 3 (1995) pp 319-401.

[Bra93] Bratsberg, S.E. “Evolution and Integration of Classes in Object-Oriented Databases”. PhD Thesis, Norwegian Institute of Technology (1993).

[CC83] Ceri, S. & Crespi-Reghizzi, S. “Relational Data Bases in the Design of Program Construction Systems”. ACM SIGPLAN Notices 18, 11 (1983) pp 34-44.

[CL93] Chiu, S.-Y. & Levin, R. “The Vesta Repository: A File System Extension for Software Development”. DEC Systems Research Center Technical Report 106 (1993).

[Con90] Connor, R.C.H. “Types and Polymorphism in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews. Technical Report CS/91/3 (1990).

[DCC93] Dearle, A., Cutts, Q.I. & Connor, R.C.H. “Using Persistence to Support Incremental System Construction”. Journal of Microprocessors and Microprogramming 17, 3 (1993) pp 161-171.

[Fel79] Feldman, S.I. “Make – A Program for Maintaining Computer Programs”. Software – Practice and Experience 9 (1979) pp 255-265.

[Jor96] Jordan, M. “Early Experiences with Persistent Java™”. In Proc. 1st International Workshop on Persistence for Java, Glasgow (1996).

[JV95] Jordan, M. & Van De Vanter, M.L. “Software Configuration Management in an Object Oriented Database”. In Proc. USENIX Conference on Object-Oriented Technologies (COOTS), Monterey, California (1995).

[Kad92] Kadia, R. “Issues Encountered in Building a Flexible Software Development Environment”. ACM Software Engineering Notes 17, 5. Proc. 5th ACM SIGSOFT Symposium on Software Development Environments, Tyson’s Corner, Virginia (1992) pp 169-180.

[KBC+96] Kirby, G.N.C., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Dunstan, V.S., Morrison, R. & Munro, D.S. “Napier88 Standard Library Reference Manual (Release 2.2.1)”. University of St Andrews (1996).

[LM93] Levin, R. & McJones, P.R. “The Vesta Approach to Precise Configuration of Large Software Systems”. DEC Systems Research Center Technical Report 105 (1993).

[LM93] Long, F. & Morris, E. “An Overview of PCTE: A Basis for a Portable Common Tool Environment”. Carnegie Mellon University Technical Report CMU/SEI-93-TR-1 (1993).

[MCC+95] 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.

[MS93] Monk, S.R. & Sommerville, I. “Schema Evolution in OODBs using Class Versioning”. ACM SIGMOD Record 22, 3 (1993) pp 16-22.

[Odb95] Odberg, E. “MultiPerspectives: Object Evolution and Schema Modification Management for Object-Oriented Databases”. PhD Thesis, Norwegian Institute of Technology (1995).

[PL83] Powell, M.L. & Linton, M.A. “A Database Model of Debugging”. ACM Software Engineering Notes 8, 4. Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, Pacific Grove, California (1983) pp 67-70.

[Roc75] Rochkind, M.J. “The Source Code Control System”. IEEE Transactions on Software Engineering SE-1, 4 (1975) pp 364-370.

[SZH85] Swinehart, D.C., Zellweger, P.T. & Hagmann, R.B. “The Structure of Cedar”. In Proc. ACM SIGPLAN Symposium on Programming Languages and Programming Environments (1985) pp 230-244.

[Tic85] Tichy, W.F. “RCS - A System for Version Control”. Software – Practice and Experience 15, 7 (1985) pp 637-654.