USING PERSISTENCE TECHNOLOGY TO CONTROL SCHEMA EVOLUTION

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

University of St Andrews, Scotland

Key words and phrases: schema evolution, persistence, referential integrity, integrated environment, hyperprogramming

Abstract

Traditional database technology may be extended by taking advantage of the facilities of an integrated persistent programming environment. This paper focuses on how such an environment may be used to provide new solutions to a long standing problem in traditional databases, that of schema evolution. A general mechanism is first described, followed by a description of a specific schema editing tool.

The persistent environment provides an underlying technology which allows the schema editor to locate and change, either manually or automatically, all affected program and data. The advantages of the mechanism are that it provides understandable semantics for evolution by controlling when the changes are made and by ensuring that changes to schema, program and data are consistent and made in lock step. It is shown how these changes together may be grouped as a transaction within a live system; furthermore, the accommodation of lazy data changes allows minimum loss of availability.

Introduction

In an orthogonally persistent programming system, the manner in which data is manipulated is independent of its persistence. The same mechanisms operate on both short-term and long-term data, avoiding the traditional need for separate systems to control access to data of different degrees of longevity. Thus data may remain under the control of a single persistent programming system for its entire lifetime. The benefits of orthogonal persistence have been described extensively in the literature [1, 2, 9, 14]. 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.

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 [2, 4]. As a result a number of persistent systems have been developed including PS-algol [18], Napier88 [15], Galileo [3], TI Persistent Memory System [21], Amber [8], Trellis/Owl [19] and Tycoon [17]. 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 [13, 14]. Object-Oriented Database Systems such as GemStone [6] and O_2 [5] have at their core a persistent object store. Process modelling systems use a persistent base to preserve their modelling activities over execution sessions [7].

Recently, integrated persistent programming systems have been developed that allow the complete software and data process to take place entirely within the persistent environment [11, 16]. The facilities provided by traditional databases may be extended by taking advantage of such an environment. In particular, this paper focuses on how the persistent environment may be used to provide new solutions to a long standing problem in traditional databases, that of schema evolution. A general mechanism is first described, followed by a description of a specific schema editing tool. The advantage of the mechanism is that it provides understandable semantics for evolution by controlling when the changes are made and by ensuring that changes to schema, program and data are consistent and made in lock step.

Evolution

Databases become obsolete when they can no longer meet the changing needs of the applications that they support. Evolution is inevitable in a long running system as the people who use the data, the data and the uses to which the data is put all change. This is reflected within the database system by changes to the data, the programs which use the data and the meta-data. Changes to program and data with the invariant of fixed meta-data are normally handled by updates to program libraries and the database respectively. The difficult problem is to change the meta-data while keeping all the existing programs and data consistent with the semantics of the change.

As an example of meta-data evolution consider Figure 1 which describes a partial schema for a parts/suppliers database. A supplier, represented by the type *Supplier*, has a name *s_name* and address *s_address*. A part, represented by the type *Part*, has a part name *p_name*, a part number *p_no* and a set of suppliers *suppliers*. There are two entries in the schema, the set of parts *PARTS* and the set of suppliers *SUPPLIERS*. This schema is designed to model each part as having a set of suppliers.

```
type address is ...

type Supplier is structure( s_name : string : s_address : address )
type Part is structure( p_name : string : p_no : int : suppliers : set( Supplier ) )

entry PARTS is set( Part )
entry SUPPLIERS is set( Supplier )
```

Figure 1. A partial schema: parts have suppliers

As the system evolves it may become appropriate to change the organisation of the data to model each supplier as supplying a set of parts. This could be represented by the schema shown in Figure 2.

```
type address is ...

type Part is structure( p_name : string ; p_no : int )
type Supplier is structure(     s_name : string : s_address : address :
                                supplied_parts : set[ Part ] )

entry PARTS is set[ Part ]
entry SUPPLIERS is set[ Supplier ]
```

Figure 2. A partial schema: suppliers supply parts

The databases represented in Figures 1 and 2 contain the same information. However, the data has different structure, the programs that operate over the data are different and the meta-data is different. The challenges in performing this meta-data evolution are:

- finding understandable semantics for the change;
- controlling when the changes are made; and
- ensuring that the changes to the meta-data, programs and data are made consistently and in lock step.

This paper demonstrates a new mechanism for tackling these challenges within an integrated persistent environment.

An Integrated Persistent Environment

An integrated persistent environment is self supporting in that it allows all the activities of manipulating program, data and meta-data to be provided by the same mechanisms. The advantages of integrated persistent environments are described elsewhere [10, 11]. For our purposes, that is controlling schema evolution, the persistent environment must:

- allow programs to be manipulated as data;
- provide an incremental loader; and
- guarantee referential integrity, which 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.

The incremental loader and the guarantee of referential integrity provide the basic facilities that allow a new kind of program representation called the *hyper-program*.

Traditionally programs are represented as linear sequences of text. Where a program accesses a database object during its execution, it contains a textual description of that object, describing how to locate the object. At some stage 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 when the program is written.

In an integrated persistent environment, 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 when the program is composed. In this case links to the objects can be included in the program instead of textual descriptions. By analogy with hyper-text, a program containing both text and links to objects is called a hyper-program. Various benefits of hyper-program technology are detailed in [12]. The most important benefit in the context of this paper is that it provides a mechanism for representing all executable programs, since free variables in closures may be represented as links within hyper-programs. This means that the system may enforce associations between all executable programs in the environment and their corresponding hyper-program source representations.

An example of a hyper-program is shown in the lower part of Figure 3. The hyper-program is a query on the parts/suppliers database to find the nearest supplier for a particular part. The schema is that of Figure 1. The links embedded in the query are denoted by rounded rectangles.

It should be noticed that the integrated persistent environment contains the data, the meta-data (schema) and the programs (queries) of the traditional database. The referential integrity of the persistent store guarantees secure links among these entities. Tools for browsing programs and the schema are designed to take into account the presence of these links.

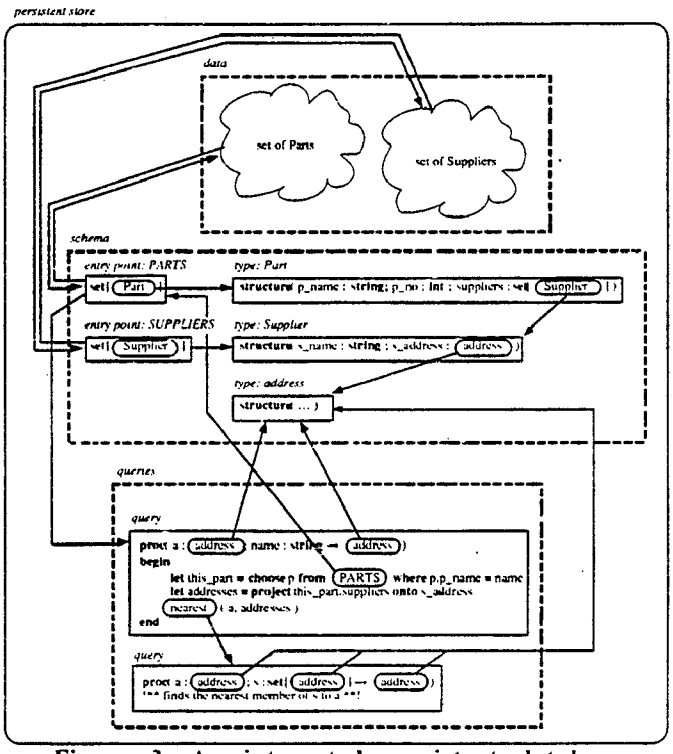

Figure 3. An integrated persistent database

Schema evolution can also exploit one of the advantages of hyper-programming, 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. 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. The conceptual simplification given by the provision of a single uniform representation throughout the life of a program is called *hyper-code*.

Since a hyper-code program representation is itself an object it may contain links, perhaps hidden to the user, to any compiled or executable form. The compiler may also arrange for the schema to record information about which programs use which data, via links. This provides the basic information necessary to find programs which access an evolving part of the schema definition.

The hyper-code abstraction hides 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; as such they are completely hidden from the programmer.

A Mechanism for Schema Evolution

The essence of this paper is to demonstrate that by utilising the persistent environment there are new solutions to the problem of schema editing. Editing the schema requires location and translation of affected queries and data. The essential elements are at hand in the hyper-code system. The schema may keep a record of which programs (queries) and data are associated with particular parts of the schema via secure links. The programs always have hyper-code source and therefore source code and data translation is possible.

The schema evolution mechanism transforms the programs and data affected by a schema edit. This is achieved as follows:

- Locate, from the schema, all affected programs and data.
- For each program which may be affected, obtain its hyper-code.
- Locate the points in the hyper-code which access the changed part of the schema and edit the hyper-code to reflect the new logical schema structure. This will involve establishing new links both to and from the changed part of the schema.
- Update the old program with the new one.
- Update the affected data with new versions. Some possible strategies for performing this are discussed in Section 5.2.

The extent to which this process can be automated depends upon the complexity of the schema change incurred. The essential point is that all interrogation and manipulation of schema, program and data occurs within a single integrated environment, and may therefore be represented as a meta-level program within that environment.

The mechanism described here relies heavily upon the self-contained nature of the persistent environment. As all the data and code is held in the same environment as the schema, it is possible to keep not only links from the schema to the data it describes but also reverse links from the schema to programs which bind to particular points of it. The hyper-code concept makes it possible to map between executable and source representations. The fact that these representations are themselves values within the persistent environment, along with the provision of a compiler in the same environment, makes this strategy possible. The next section describes a schema editing tool which exploits this mechanism.

The Schema Editing Tool

The schema editing tool allows the description and revision of the schema through a graphical user interface. Unlike normal schema editors, however, it also gives its user help in handling the consequences of the schema change, by locating all affected programs and data and allowing their revision to be partly or even fully automated.

Figure 4 shows two windows provided by the schema editing tool. The first contains a view of the schema which allows the schema to be edited interactively. It can also be used to aid the construction of queries. An example query is shown in the second window: it calculates the address of the nearest supplier of a particular part, using the schema of Figure 1. The query contains links to the schema for *address* and *PARTS*, created by mouse gesture over the schema diagram. Any change to these parts of the schema may thus require the query to be changed. Not shown in the diagram are the links from the schema entities to the queries. Also not shown is the function *nearest* which is hyper-code already in the persistent environment.

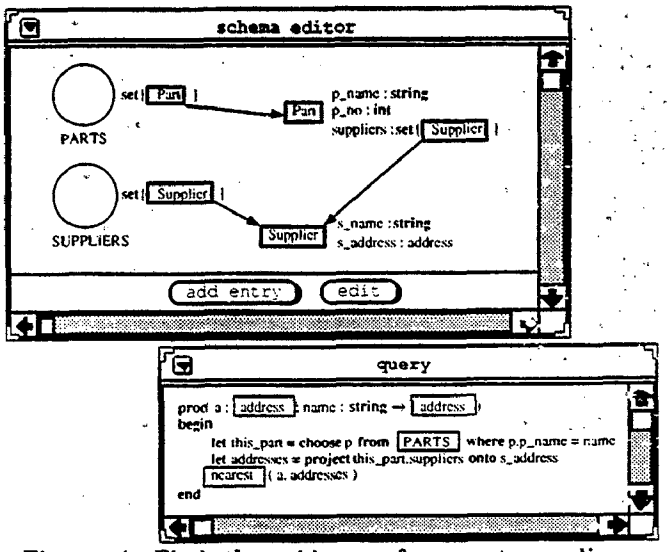

Figure 4. Find the address of nearest supplier—parts have suppliers

The placing of the schema entry *PARTS* into the query causes the query to be inserted into a table maintained by the schema editor. This may occur when the query is edited or compiled, depending upon the implementation details of the system. When the schema is subsequently edited, and a change to the type *Part* is made, this query is one of the candidates which may require change as it is reachable through the closure of reverse links from the changed type definition.

More sophisticated information may be derived from static analysis of the query; in this case, for example, the query accesses only *p_name* and *suppliers* fields of values of type *Part*. If a change to the schema does not affect these fields there is no need to change this query.

The required change is now made to the example schema. This is achieved by invoking the schema editor and making the changes as shown in Figure 5. Once the change has been made the tool is still active and now goes through the phase of locating affected programs and data. Firstly it deals with the programs.

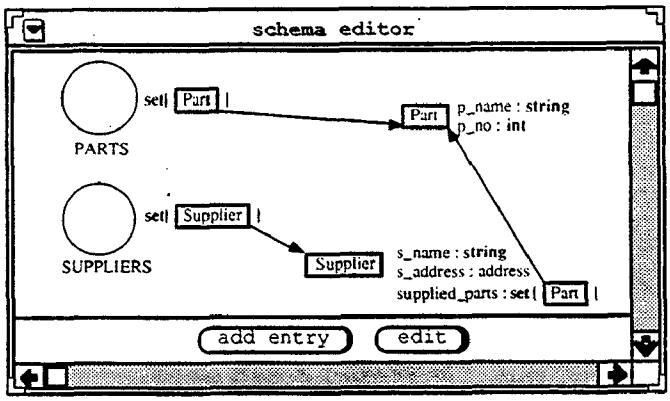

Figure 5. Changing the schema

Changing Affected Programs and Queries

The least to be expected is that each query which accesses data of type *Part* is presented to the user for editing. However, in some cases it may be possible to do better than this. The provision of source representations and the compiler all within the integrated persistent environment means that instead of

each program being manually changed, a change may be specified once and applied to each program automatically.

The dialogue shown in Figure 6 could be made with the schema editing tool.. As stressed before, the importance of the persistent environment is that the technology exists to program automated changes from such a user interface.

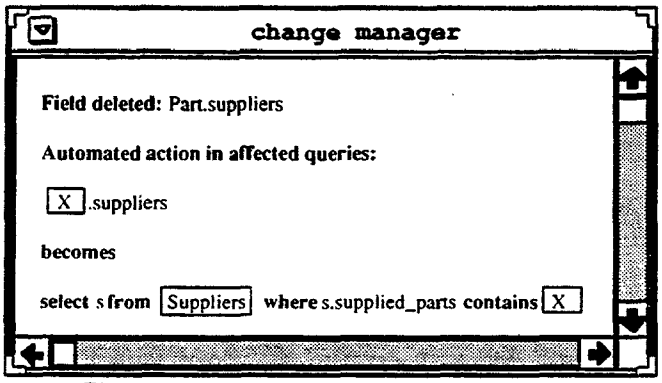

Figure 6. Automating program change

This kind of automation could be dangerous in general, as can any global search/replace facility; the programmer can specify options such as show individual occurrences, global replacement, etc. One danger, for example, is that inefficient code can result. Even so, the ability to automatically provide at least working code is a considerable achievement. Of course the semantic equivalence of the search/replace expressions can not in general be decided automatically. Figure 7 shows the new version of the query after being transformed:

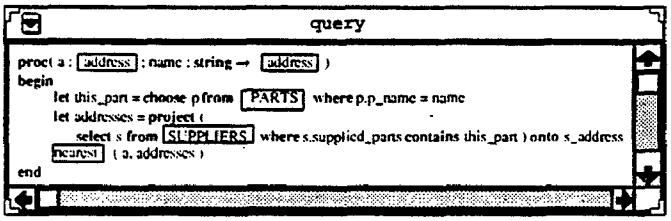

Figure 7. Find the address of nearest supplier—suppliers supply parts

There are a number of further dimensions to the process of automatic query change that are not seen in this example. For example, the automated change indicated will work correctly only when the *suppliers* field is used as an R-value; programs which update the *suppliers* field must be handled quite differently. This different usage can of course be detected by the compiler and different automated changes would need to be provided for the different cases.

Automating Changes to Affected Data

Continuing with the example, the affected data must also be changed to match the new schema.

The removal of the *suppliers* field from type *Part* causes no real problems. The only purpose of physically removing the field from the data is for efficiency, to avoid wasted space. For the logical schema to be honoured it is necessary only that no programs may access the old field, and that correct field addressing is maintained by all programs using values of type *Part*. The associations of the compiler, the schema definition and the executable code make these relatively straightforward to manage, including the physical reorganisation of the data if required. It may be imagined that such reorganisation could be automated to take place during quiescent periods in the system.

The addition of the *supplied_parts* field to the *Supplier* type requires logical intervention by a programmer to specify initial values for the reformatted data. Once again, however, the architecture can be used to ease the initial value problem by supplying a flexible range of possibilities. In the case of the example, the initial values may be gleaned from the existing database using the initial value dialogue as shown in Figure 8. Once again, the description of the schema using the hyper-code principles allows the identification within the schema of all instances of the changed type.

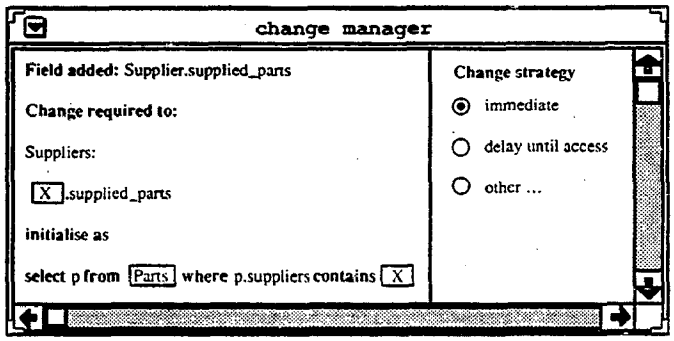

Figure 8. Automating initial values

The initialisation code is expressed in terms of an expression which, in this case, is resolved according to the previous version of the schema. At this point the change to the schema has not yet been committed; this dialogue is part of the schema editing process, and the old data is therefore still available.

Notice also that the initial value may be specified to be evaluated immediately, when the individual new fields are accessed, or with some flexible arrangement between these two extremes. The tradeoffs among these are well documented elsewhere; the point here is that the change management tool is sufficiently flexible to allow for the creation of initial values by any computable expression (including for example by querying the user) and at a flexible range of times. It is even possible for the association between the compiler and the schema description to be used to allow expressions which access the previous schema description, as in this example, to be executed lazily at some point in the future. This is because the arrangement for the physical deletion of obsolete data may be left within the control of the same integrated system.

Committing the Schema Change

The change to the example schema is now complete. Changes have been specified to the schema itself, and to all programs and data which are affected by the schema change. These changes are viewed as an atomic transaction: either all the specified changes must be committed atomically or they must all be discarded. The important point is that the problem of programs and data being inconsistent with the changed schema has been eliminated, as all such programs and data can be detected, and the schema change can be automatically disallowed if appropriate matching changes have not been specified.

Kinds of Schema Evolution

So far the only example of schema change given has been one where the same net semantic data is modelled in the schema before and after the change. To analyse the consequences of different kinds of change, the following categories of schema change are identified:

- additive — extra semantic knowledge is modelled
- subtractive — less semantic knowledge is modelled
- descriptive — the same semantic knowledge is modelled in a different manner

The necessary effects of these kinds of change on programs and data is now examined.

Additive Evolution

There is a temptation to imagine that additive changes have no required effect on existing programs, and require only initial values to be entered into the database. This is not quite true, however, as many queries are of a form which require the presentation to the user of all known, or at least all available, information about a data item. Therefore although all programs that exist before the change will continue to execute correctly in the mechanical sense, their intended semantics may require them to be changed. An example of this is a query that writes out all the fields of a *Supplier*.

This aggravates the problem of identifying all programs which require change, as it requires semantic knowledge which cannot be deduced from a program text. Two alternatives are possible. Firstly, all programs which access any value of a data type to which functionality has been added could be presented to the programmer for possible update. The other possibility is for this kind of program to be identified by the programmer at the time it is created, so that only these programs will be presented back to the programmer in the circumstances of additive change.

Additive change requires a corresponding change to the data. As mentioned above, there are many times at which this change can be made, so long as it occurs before the new data is accessed. This kind of change with respect to data is possibly the best documented, and requires no further elaboration here.

Subtractive Evolution

Subtractive change occurs when part of the data model becomes obsolete and is no longer required. With respect to programs, once again the query which presents all the available data about an item must be considered. Without such programs, it is reasonable to assume that any programs which are found to access the deleted entities are themselves obsolete; if this is not the case, then the schema change has been discovered to be pre-emptive and should be aborted! However a number of programs may well exist which routinely access the fields to be removed, and these may be altered simply by removing the obsolete accesses.

As already mentioned, this kind of schema change brings no required action in terms of the data except in terms of space wastage.

Descriptive Evolution

Descriptive change occurs wherever the description of the schema is changed but the semantics modelled is equivalent before and after the change. Such changes are made for convenience or efficiency; they are typically regarded as the hardest to handle in traditional database systems.

In the light of the schema evolution strategy described here, such changes are the easiest to handle. In terms of existing code, this is the only kind of change where all affected programs may be reliably detected and automatically updated to preserve their previous semantics. Similarly for data, the semantic difficulties of the initial value problem are avoided, as any initial values may always be calculated from the previous state of the schema.

The Initial Value Problem

A powerful attribute of the mechanism described is the flexibility with which the accommodation of initial values is defined. They may be the result of arbitrary computations, including those which depend upon the schema description immediately before the schema edit is defined. These computations may be programmed to occur at a range of different times.

Value access through obsolete schema definitions is achieved through the persistent base technology, where a value's persistence is defined by its reachability. Even when values are inaccessible from the schema, they will still persist for as long as the meta-level programs which implement the system can reach them. Values will be garbage collected only when the meta-level system can no longer access them.

The time of calculation of the initial value is left as flexible as possible to allow for different circumstances of user load, schema update time and so on. If initial values are calculated at the time of schema update then the system may be effectively out of action for a considerable time. This is particularly so if the specification of an initial value includes querying the user for input, which will often be the case when additive schema changes are made. In many cases it may be better to delay the initial value calculation to a more suitable time, possibly as late as when the individual new values are accessed.

The ability to delay the initial value calculation until the time of first access is made possible by the integration of the schema description mechanism and the compiler. When code which accesses a possibly uninitialised value is compiled the executable code contains a test for the initialisation of the value. This information may be kept in a central table, or a reserved value may be used. If the test fails then the system suspends the accessing process and causes the initialisation code to be executed [20].

Delaying initial value calculations to the time of access may also cause problems. Peak calculation cost may occur at the same time as peak user load, causing efficiency problems. On a semantic level, it may not be acceptable for a query made by a user to result in exactly the same query being presented back to the user! This may occur when the programmer has specified that initial values are to be obtained by querying the user. Thus a number of mixed strategies may be allowed for, in which delaying the calculation until access is used only as a catch-all safety mechanism. For initial values which do not require user interaction, the calculation may be performed at the time of low user load, or a later "down" period if this is acceptable to the usage of the database. For those which do, a separate initialisation process may be spawned for a data entry operator. Even in this case, however, it should be noticed that the database may become live immediately, by the programming of an appropriate action for the case where an access should occur before the data is entered.

The Schema Change as a Transaction

The database schema edit is a privileged operation, and users of the database should be shielded from its effects. Normally this is possible to achieve only by closing down the database while schema editing is in progress.

Since the schema is itself data in the persistent environment, the schema edit may be regarded as a transaction in its own right. The description of the schema editing changes can all be carried out in parallel with any other activity; it is only at the schema change commit time that contention may occur. If the database concurrency control mechanism is generalised to encompass access to the programs in the system, as well as the data, then the updates associated with the schema edit can be treated as a normal transaction within the system.

In cases where changes to the data are delayed to any time after the schema change, then a standard readers/writers protocol has

the effect that the schema change commit may take place at any time when none of the affected programs are actively in use.

Conclusions

This paper has outlined a mechanism by which schema evolution can be integrated with changes to all affected programs and data within the system. 'A single evolutionary step may be described as an atomic transaction within the system, thus completely eliminating the problems associated with inconsistencies between the schema, program, and data.

The ability to perform such comprehensive schema editing is made possible by the use of persistence base technology. The key point of the technology is that the representations of the schema, programs, and data are all maintained within a single persistent environment. This environment is able to maintain data structures which model the conceptual bindings between the three entities. This gives the ability for programs within the persistent environment to enforce that changes are performed in lock step.

A prototype implementation of the schema editing tool described is under way at the University of St Andrews. It is built using the language Napier88 [15], on top of the hyper-programming environment [11] and using the WIN graphical user interface package. These systems are all available from the authors; it is hoped that the schema editor will be made available some time in the near future.

Acknowledgements

The authors are grateful to Malcolm Atkinson for the benefit of his many ideas about database evolution, some of which have undoubtedly found their way into this paper. Richard Connor is supported by SERC Postdoctoral Fellowship B/91/RFH/9078.

Correspondence address:

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

email: {richard, quintin, graham, ron}@dcs.st-and.ac.uk

References

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

[2] 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.

[3] Albano, A., Cardelli, L. & Orsini, R. "Galileo: a Strongly Typed, Interactive Conceptual Language". ACM Transactions on Database Systems 10, 2 (1985) pp 230-260.

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

[5] Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lécluse, C., Pfeffer, P., Richard, P. & Valez, F. "The Design and Implementation of O_2, an Object-Oriented Database System". In **Lecture Notes in Computer Science 334**, Dittrich, K.R. (ed), Springer-Verlag (1988) pp 1-22.

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

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

[8] Cardelli, L. "Amber". AT&T Bell Labs, Murray Hill Technical Report AT7T (1985).

[9] Connor, R.C.H., Dearle, A., Morrison, R. & Brown, A.L. "Existentially Quantified Types as a Database Viewing Mechanism". In **Lecture Notes in Computer Science 416**, Bancilhon, F., Thanos, C. & Tsichritzis, D. (ed), Springer-Verlag (1990) pp 301-315.

[10] Farkas, A.M., Dearle, A., Kirby, G.N.C., Cutts, Q.I., Morrison, R. & Connor, R.C.H. "Persistent Program Construction through Browsing and User Gesture with some Typing". In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag (1992) pp 376-393.

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

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

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

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

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

[16] 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).

[17] Matthes, F. & Schmidt, J.W. "Definition of the Tycoon Language TL - A Preliminary Report". University of Hamburg Technical Report 062-92 (1992).

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

[19] Schaffert, C., Cooper, T. & Wilpot, C. "Trellis Object-Based Environment Language Reference Manual". DEC Systems Research Center Technical Report 372 (1985).

[20] Skarra, A.H. & Zdonik, S.B. "Type Evolution in an Object-Oriented Database". In **Research Directions in Object-Oriented Programming**, Shriver, B. & Wegner, P. (ed), MIT Press (1987) pp 393-415.

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