This paper should be referenced as:

Kirby, G.N.C., Morrison, R., Connor, R.C.H. & Zdonik, S.B. “Evolving Database Systems: A Persistent View”. University of St Andrews Technical Report CS/97/5 (1997).

Evolving Database Systems: A Persistent View

G.N.C. Kirby, R. Morrison, R.C.H. Connor* & S.B. Zdonik†

School of Mathematical and Computational Sciences,
University of St Andrews, St Andrews, Fife, Scotland.

*Department of Computing Science,
University of Glasgow, Lilybank Gardens, Glasgow, Scotland.

†Department of Computer Science,
Brown University, Providence, Rhode Island, USA.

{graham, ron, richard}@dcs.st-and.ac.uk
sbz@cs.brown.edu

Abstract

Orthogonal persistence ensures that information will exist for as long as it is useful, for which it must have the ability to evolve with the growing needs of the application systems that use it. This may involve evolution of the data, meta-data, programs and applications, as well as the users' perception of what the information models. The need for evolution has been well recognised in the traditional (data processing) database community and the cost of failing to evolve can be gauged by the resources being invested in interfacing with legacy systems.

Zdonik has identified new classes of application, such as scientific, financial and hypermedia, that require new approaches to evolution. These applications are characterised by their need to store large amounts of data whose structure must evolve as it is discovered by the applications that use it. This requires that the data be mapped dynamically to an evolving schema. Here, we discuss the problems of evolution in these new classes of application within an orthogonally persistent environment and outline some approaches to these problems.

Keywords: persistence, views, schema evolution, unstructured data

1 Introduction

The growing requirements of database application systems challenge database architects to provide the appropriate mechanisms for system evolution. Traditional database systems are designed under a number of *a priori* assumptions about how they will be used that fundamentally affect their ability to evolve. These include:

- **Order** - The database is constructed and used in strict order by defining the meta-data (schema) first and then initialising some data in accordance with the meta-data description. Programs and queries may then use the data and when necessary generate more data. Where data with a new structure is required the process is repeated in the same order. Thus programs that discover new structure about existing data have an ordering difficulty. Before they run, they must ensure that the schema contains the new structure which is impossible since the new structure is only discovered during execution.

- **Content** - The information in the database is contained in three separate parts: the schema, the programs and the data. The mechanisms for using these parts are different and their independent use can lead to inconsistency in addition to the complexity of non-uniformity. This increases the difficulty of discovering new structure and applying change consistently.
- **Fixed name space** - The values and objects in the database are identified by their names which are contained in the schema and fixed for the lifetime of the database. This makes it difficult to evolve the schema, merge databases with duplicate names and combine applications that may have name clashes.
- **Size** - The size of the data is expected to be large in comparison with the size of the meta-data, hence the term bulk data. While this is true of some traditional relational systems, where the structure of the data is first-order and relatively simple, it is less so in object-oriented databases and persistent object systems which support complex objects. The trade-off between the size of the data and the meta-data alters the cost of propagating evolutionary changes.
- **Completeness** - The description of the data is complete and all the data and meta-data is consistent and correct. As new structure is discovered in the data, meta-data and programs that use the data only partial completeness can be assumed.
- **Structure** - The schema defines the storage structure of the data and its location within the database. Data of a single type is stored in a finite (small) number of fixed forms, usually constrained by size, for efficient storage. This does not accommodate the need for new storage formats to efficiently implement new data structures.

This paper focuses on new classes of application, such as scientific, financial and hypermedia, that require new approaches to evolution. These applications are characterised by their need to store large amounts of data whose structure must evolve as it is discovered. This requires that the data be mapped dynamically to an evolving schema, involving the freeing up of the traditional notions of order, content, fixed name space, size, completeness and structure. The problems of evolution in these new classes of application will be discussed within the context of an orthogonally persistent environment [ABC+83, AM95] and mechanisms proposed which will go some way to accommodate the evolutionary needs of data, meta-data, programs and users.

To illustrate our points, we develop a simple meteorological database where the data is collected from satellites at a rate that is too fast to analyse the structure of the data, before storage. We show how new statically predicted stucture may be imposed on this data by considering data constructors as viewing mechanisms. Where the constructor is abstract, information hiding may be achieved without encapsulation [CDM+90]. We also show how to accommodate schema change where the new structure of the data is not predicted statically but generated dynamically by some application using the data. Some of the techniques in this paper are already known. However, their combination is unique as is their application to this new set of challenges.

2 New Challenges

Zdonik [Zdo93] has identified new classes of application system that require new approaches to evolution. These include scientific applications, data mining, financial applications, multimedia applications, graphics and video applications, text applications and heterogeneous databases. The applications are characterised by their need to store large amounts of data whose structure must evolve as it is discovered. The changes may be additive, subtractive or descriptive [CCK+94a], but all require that the data be mapped in complex ways, and

dynamically, to an evolving schema. The nature of these applications may be illustrated by the example of a scientific application, taken from [Zdo93].

Consider a satellite that is sending weather maps to a monitoring station. Image enhancement techniques might be applied to the data as it arrives, but it is stored initially as a large bitmap without any higher level information concerning its contents. Further additional structure may be discovered by an application program that is able to perform feature extraction on this image and can identify and categorise various kinds of storms. For example, while the weather map is an object, each storm might best be considered as an object as well. The storms are embedded inside the weather map. The fact that this weather map contains one or more storms would be stored as a part of the weather map object which might cause it to become reclassified as an *inclement weather map*. Each storm might also contain some substructure. For example, if the storm is a hurricane, the storm object might have an eye, a size, and a location. For this to occur, the schema needs to dynamically change to suit the needs of the application.

It should be noted that there is no requirement that the objects that are discovered in the weather map be disjoint. Other feature extractors could identify weather fronts that intersect several of the storms. The embedded objects do not necessarily partition the weather map nor do they form a hierarchy. The *weather map* type might contain some constraints regarding cloud motion that is based on the normal position of the jet stream. A new weather map or series of weather maps might cause the feature extraction application to determine that the jet stream has moved to a position that would not be allowed by the constraints in the current schema. The application would then suggest a change in the schema based on this new information.

The above example illustrates many of the problems incurred by applications that use databases outside the commercial world. In summary, they pose the following new challenges to database architects:

- **Run-time schema change** - As the new structure is discovered, for example in the discovery of *hurricanes* and *weather fronts* in the *weather map*, it must be possible to incorporate that new structure into the schema while the program is running.
- **Complex mapping to the schema** - A mechanism is required that will keep track of the complex relationships amongst the data, meta-data and programs, and indeed the intended semantics defined by the users.
- **Object type migration** - A mechanism is required for descriptive evolution, as in the reclassification of the *weather map* as an *inclement weather map*, even where the new form of the data differs markedly from the old. Existing applications may operate with new data if the system supports a type system that allows new structure to be related to existing structure, such as subtyping [Car84] or mechanisms geared specifically to evolution [CBM95]. A more powerful technique is required where such a mechanism is not supported or where the evolution does not follow the predicted path provided by the type system.
- **Embedded and overlapping objects** - Encapsulation requires that objects hide their internal structure. Some objects, like the *hurricane* in the *weather map*, may, however, be part of others. To model this accurately requires a mechanism that will allow an aggregation superstructure to be placed on the object without encapsulation, while retaining the property of information hiding where necessary. Similarly two objects may overlap, such as two *weather fronts* in the *weather map*. In both cases, unlike in traditional object models, multiple objects share state and therefore pose problems of consistent update.

- **Co-ordinate systems** - Applications may require more than one co-ordinate system to be active at one time. For example, the *weather map* may have its own co-ordinate system which is active at the same time as the co-ordinate system required by the *hurricane* object. The system should provide and support the translation between these systems.
- **Non-contiguous object specification** - Conceptually objects in a database system are regarded as if they occupy contiguous storage. Where the data contained in a new object is non-contiguous, such as alternate rows of a matrix, a mechanism is required to group the parts into an apparently contiguous object with separate identity.
- **Breaking abstraction** - Hiding data in objects as they are formed runs the risk of improperly imposing structure on data. As we have seen, data may require several forms thus it must always be possible to access the original form, subject to appropriate authorisation.

The following sections will consider database applications based on a persistent programming system. Nonetheless, a traditional schema will be assumed to locate the data. The schema, which may be distributed, contains the types of the data and entry points for both code and data. Each entry point can be considered as an application or a data object.

3 The Persistent Environment

The contribution of orthogonal persistence [ABC+83, AM95] is to integrate the notions of long and short term data and to allow programs to be treated as first class data objects [AM85]. The facilities of orthogonally persistent programming systems, such as Napier88 [MBC+94], Tycoon [MM93] and Fibonacci [AGO94] may be used to address the evolutionary problems described earlier. The features that are required include a persistent store that contains data, programs and the meta-data. In strongly typed systems, such as these, the meta-data consists of type assertions about the contents of the persistent store; these are analogous to a traditional schema [AM95].

Applications which discover new structure, such as a data mining program, may compute over the data, programs and meta-data structure. Orthogonally persistent systems allow the limitations of traditional systems in dealing with evolution, as outlined in the introduction, to be eased in the following manner:

- **Order** - The essential property is to allow schema update during execution. One such mechanism is linguistic reflection [SMK+93, SSS+92] which allows computations over the meta-data representations, and new data, programs and meta-data to be bound into the executing system.
- **Content** - The complete database, i.e. the schema, the programs and the data, is contained in a single persistent object store. The mechanisms for using these parts are uniform allowing new structure to be discovered and utilised in a uniform manner. Thus applications may compute over existing data, programs and meta-data in a uniform manner and in one environment.
- **Fixed name space** - All objects in the persistent store are anonymous and have unique identities. Names are associated with access paths rather than objects. Applications may thus impose their own name spaces on the data, allowing the same object to be used in different applications with different names.

The evolution of orthogonally persistent systems will be illustrated, with reference to the meteorological database, in two parts: mechanisms for imposing multiple layers of structure on the data, and mechanisms for evolving meta-data. The concept of all data constructors as viewing mechanisms [CDM+90] will be used for structuring data, dealing with the new

problems of embedded objects, co-ordinate systems, overlapping objects, non-contiguous object specification and breaking abstraction. Linguistic reflection and hyper-programming [KCC+92] will be used for evolving the meta-data including run-time schema change, object type migration and complex mapping to the schema. Persistent programming systems may also be used to address the problems of size, completeness and structure mentioned in the introduction, but space forbids development at this time.

The mechanisms described here may be combined and used under a single methodology. Indeed it is this combination in the presence of a single uniform persistent store that makes the approach unique.

4 Structuring Mechanisms

4.1. Multiple Views

The methodology used to accommodate evolution in data and programs is to regard all data constructors as views over the data. Initially data is placed in the persistent store in the manner in which it is collected or generated, which will then be regarded as its most primitive form. Systems are built in terms of multiple views of the primitive data and may involve many layers of views. The hypothesis is that each view may be described by a consistent set of view mechanisms and that change control across a view boundary may be achieved using a single methodology.

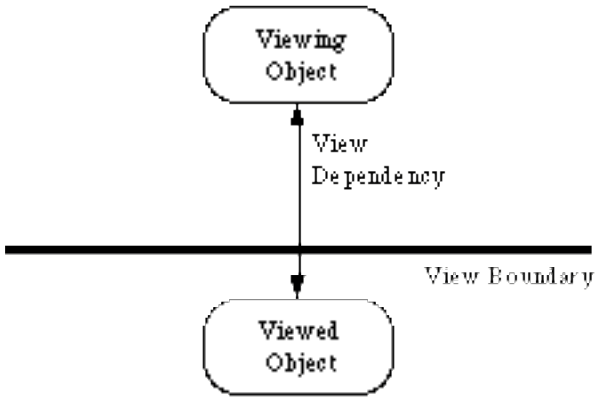

Figure 1. View Dependencies

Figure 1 depicts the relationship between two objects across a view boundary. The view boundary is, in general, bi-lateral and many-to-many since the viewing object may view many objects across different boundaries and the viewed object may be multiply viewed across many view boundaries.

Views may be open, and defined implicitly by the data constructor, or abstract and defined explicitly by a set of functional interfaces. Thus a view forms a functional dependency over an existing object. Every object in the system is defined by a signature, consisting of a set of operations available over the value and an implementation consisting of the operations. In the abstract case the definition of the signatures and the implementations are divorced allowing different implementations to be used at different times. Signatures are, however, fixed for the lifetime of an object (view). Co-ordinates, vectors and records, for example, may be used to provide open views and existentially quantified types to provide abstract views.

There is an obligation to maintain the functional dependencies in such a way that changes are reflected through the viewing interface. Where all the views are abstract, and provided by

explicit functional dependencies, it is a simple matter to maintain the consistency between views at different levels of abstraction. In the weather map example, the storm information may be stored initially as a co-ordinate system within the weather map. The storm view may be written in terms of functions which operate over the weather map, consisting of the translation rules for mapping from the weather map to the storm and in reverse. The functions guarantee the consistency of the use of the views under some concurrency control for update.

The solution may however be clumsy especially if the storm is described in terms of the same co-ordinate system as the weather map. In this case the functional dependencies must be defined when the implicit view is created in order to maintain consistency. The discussion highlights the need for mixing open and abstract views in a manner that allows the user to manipulate the appropriate abstraction. It is a problem that will not, however, be addressed in this paper.

4.2. Implementation Issues

Consider again the meteorological database. Initially the data is held in an unstructured form, just a collection of photographic images captured by various satellites. This data is then used by a weather forecasting application to build higher level models of weather patterns, and ultimately to produce forecasts. Due to space limitations the example here will consider only the first stage of analysis and structure formation. This involves running a feature extractor application over the photographic data. The purpose of this application is to use image analysis to determine the locations of static features, such as landmasses and oceans, and dynamic features such as fronts, storms and hurricanes. Thus as the program is running it discovers new structure in the original data and forms new views over it.

Figure 2 shows the database schema, which contains three entry points, a set of type definitions, and an application interface. The entry points refer to sets of objects of the corresponding types.

```
entry MAPS is     set [WeatherMap]
entry FEATURES is set [Feature]
entry AREAS is    set [Area]

type WeatherMap is   structure (satellitePic : Bitmap;
                                features : set [Feature])
type Bitmap is …     ! arbitrarily shaped region of pixels;
                     ! storage structure not specified here
type Feature is      structure (featureType : string; featureArea : Area)
type Area is         structure (mapRegion : Region; picture : Bitmap)
type Region is …     ! representation of region in real-world coordinates

application extractFeatures : proc (WeatherMap)
```

Figure 2. Database Schema

As each raw photographic image arrives it is stored initially as a *WeatherMap* object with an empty feature set, and a reference to it entered in the entry point set *MAPS*. Later, when the feature extraction application is run against the image it creates additional views comprising *Feature* and *Area* structure objects which contain references to the raw data. These objects are entered in the *FEATURES* and *AREAS* entry point sets; subsequently users can access the data via the views provided by any of the three entry points. A simplified diagram of this structure is shown in Figure 3:

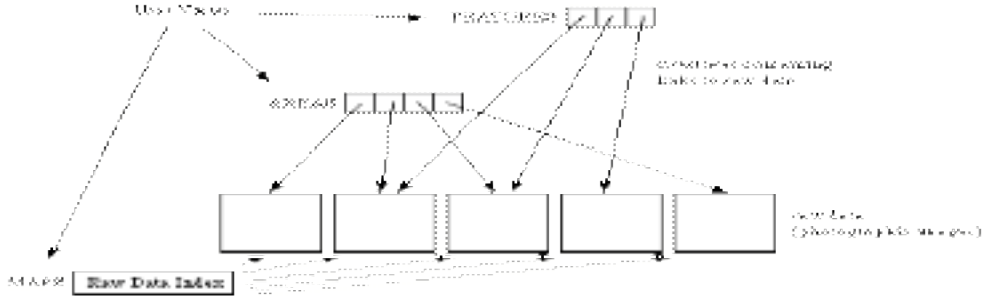

Figure 3. Structure Views with General Access

Here the raw data is partitioned into chunks and accessed by users through views provided by structures which contain links to the raw data. The perceived ordering of the raw data may vary depending on the view. There is nothing to stop users accessing the raw data directly if they wish, via the index which contains links to all the raw data chunks. This access may be restricted to the administrator by imposing password protection on the access to the raw data as shown in Figure 4. Users may now access only those parts of the raw data allowed by their views; since user address arithmetic is forbidden there is no way to access one chunk directly from another. The administrator may gain access to the index and thus the raw data by presenting the correct password to the checking procedure which contains a link to the index encapsulated within its closure.

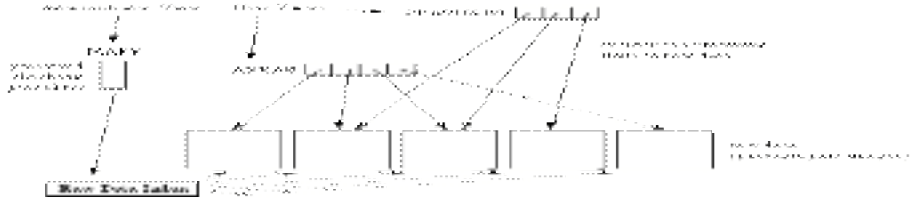

Figure 4. Structure Views with Restricted Access

Procedural encapsulation can be extended further so that users have no access to the raw data itself. Instead a user view contains procedures which contain encapsulated links to the raw data; the user is restricted to the functionality provided by the procedures, as shown in Figure 5:

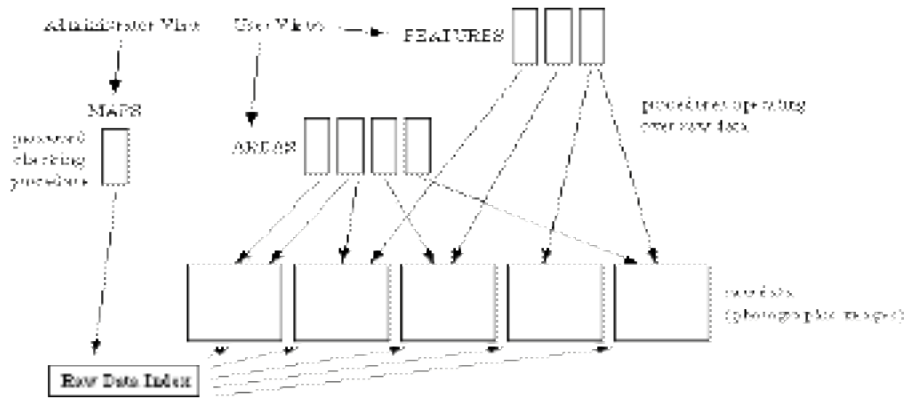

Figure 5. Encapsulated Views

This mechanism prevents users from accessing the raw data at all. Another possibility is to use abstract data types to provide user views, allowing users to access parts of the raw data directly but with limited type information. This restricts the operations a user program may

perform on the raw data, while retaining the ability to pass references to the raw data to interface procedures [CDM+90]. For example, the database may implement a feature object as a structure containing two address mapping functions as described earlier, and provide a procedure in a user view which creates a new feature object. That procedure returns to the user a reference to the structure implementing the object, with a restricted type. The user cannot discover the contents of the structure, or even that it is a structure. All they can do with the reference is to pass it to other interface procedures which operate on weather features.

Notice that these viewing mechanisms can support embedded and overlapping objects. For example, *Area* objects are embedded within the original *WeatherMap* objects, and may overlap other *Area* objects. For example, the Napier88 language supports images as a basic co-ordinate system of pixels in an infinite two dimensional integer space. If one of the photographic images is modelled by a Napier88 image type, then views to part of the image may be formed by the limit operation. For example

let firstView = **limit** photoImage1 **to** 200 **by** 300 **at** 20, 20

let secondView = **limit** photoImage1 **to** 1000, 1200 **at** 50, 50

All images have their own origin (0, 0). *firstView* is that part of *photoImage1* that starts at the pixel 20, 20 and has the size 200 x 300. Translation to the correct origin is automatic in the context of the view. Notice that the two views overlap and are embedded in the *photoImage1* views. Other embedded and overlapping objects may be formed by the same object being in more than one view. Notice also that, although not demonstrated here, views may be formed over other views to any depth and any mixing of levels.

It is interesting to compare this style of viewing with encapsulation and information hiding in object oriented database systems. In the latter the raw data may only be viewed through one interface and the information is essentially trapped in the object once instantiated. In this technique, the data is placed in an object (view) dynamically when the data modelling requires it. The viewed value and the viewing value are both available to other views of the data and there is no sense in which the viewed value becomes unavailable, or encapsulated, in the viewing object. Of course, it is not always desirable to expose all views of data and techniques are required to limit the visibility of certain data. However this is achieved by information hiding and not encapsulation and thus overcomes any conceptual difficulties in breaking encapsulation even when the views are abstract.

Once a new view has been established it must be placed in the schema. So far the new views created have been predicted in advance by the feature extraction application and can therefore have entries and type defined in the schema before the application executes. Placing the new view in the schema involves altering data reachable from some entry point or establishing a new entry point.

5 Meta-Data Evolution

We now show how a completely new view, i.e. one that is not statically expressed, may be placed in the schema of the meteorological database. The application itself may change the schema dynamically, which involves the following steps:

- the need for a schema alteration is identified by the application, such as the establishment of a new view;

- the application obtains a representation of the current schema, constructs the necessary schema change and modifies the representation appropriately;
- the modified representation is checked for consistency and a new schema derived from it;
- all data objects and applications affected by the schema change are updated to make them consistent; and
- finally, execution can continue.

Note that if the application needs to continue executing after the schema change then it must itself be updated to be consistent with the new schema. As with the schema change, updates to objects and programs are achieved by creating new versions which are then substituted. If all of these operations are performed within a transaction then the schema evolution may take place within a live system in which there are other applications running.

5.1. Mechanisms

Supporting schema evolution requires that all the inter-dependent parts of the schema, data and applications are changed consistently with the schema change. The advantage of basing the database on a persistent system is that since the schema representation, data and applications are all held as objects in a persistent store, each can contain links to the parts on which they depend and vice-versa. Further, the integrity of these links is guaranteed through the store's property of referential integrity, so it will always be possible to follow the links to find, for example, all of the applications which depend on a particular schema entry point [MCC+95].

Once affected data and applications are located they must be updated to make them consistent with the new schema. This can be achieved by creating new versions and substituting them for the existing versions. The feasibility of automating this process depends on the complexity of the schema change, and on whether the change is additive, subtractive or descriptive [CCK+94a].

5.2. Dynamic Evolution of Meteorological Schema

Figure 6 shows a simplified version of the schema of Figure 2. Here only the photographic images are stored, in a simple format which allows only rectangular bitmaps. The next section will then show how this may be evolved dynamically to support the definition of views which were not predicted statically.

```
entry MAPS is set [WeatherMap]

type WeatherMap is structure (satellitePic : image)

application extractFeatures : proc (WeatherMap)
```

Figure 6. Initial Schema Definition

In order to support future evolution, the schema, data and applications are all represented as typed objects in the persistent store, as shown in Figure 7. The schema representation is a graph of objects which together describe the type and entry point definitions. For each entry point the database stores links to the associated data objects and to the definitions of the applications which refer to it. Similarly, the data objects and application definitions are stored together with reverse links to the corresponding entry point and type definitions. This is

possible for the application definitions since they are stored as hyper-programs [KCC+92] which contain direct links to other persistent objects. This will allow all affected parts of the database to be located should a schema change become necessary.

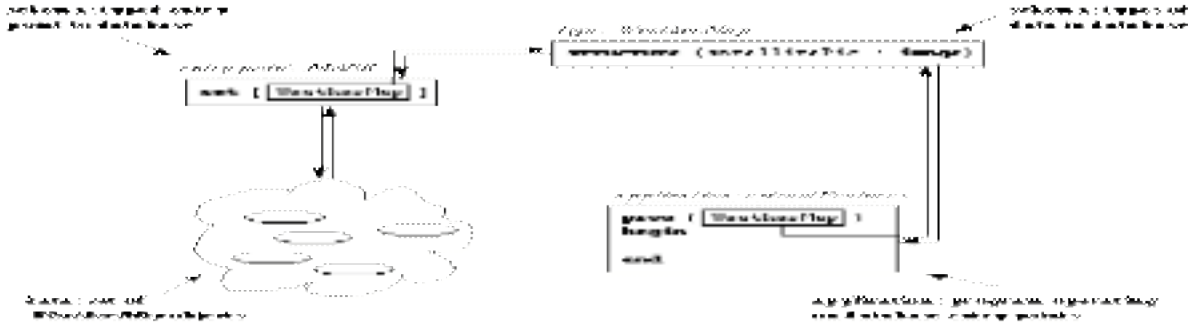

Figure 7. Schema, Data and Programs as Persistent Objects

The feature extraction application is now run over the simple initial database, and on one particular image locates the static and dynamic features shown in Figure 8:

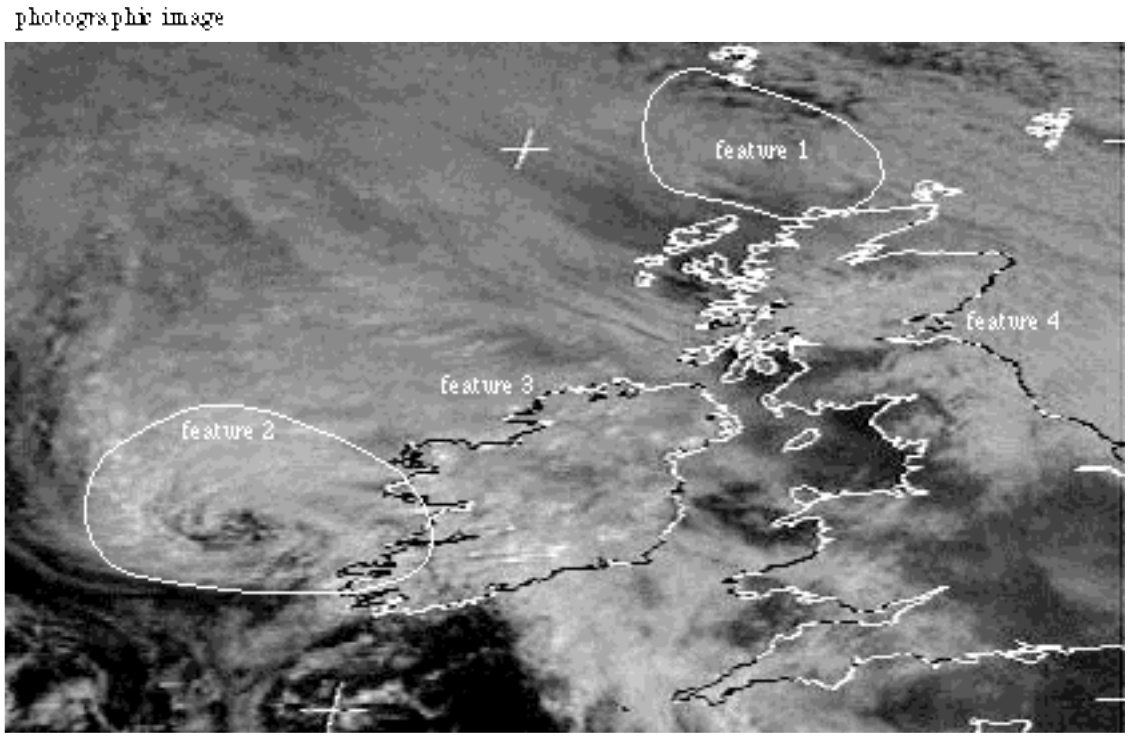

Figure 8. Features Located in a Weather Map

In order to store the derived information about the features, the application now refines, dynamically, the schema to that shown originally in Figure 2. This refinement is a combination of an additive change in which the field *features* is added to *WeatherMap*, and a descriptive change in which the type of the field *satellitePic* is refined to *Bitmap*.

The incorporation of the new type definitions into the schema is achieved through linguistic reflection. Having determined the required new types, the application generates a source code representation of the types, for example strings or hyper-code, and then transforms that representation into a table which maps type names to system-level type representations. This

requires the presence of a compiler as a procedure in the database. The application then calls system procedures to delete the obsolete entry point and to create the new ones. Figure 9 shows the dynamically generated source representation of the new types:

```
type Region is …
type Bitmap is …
type Area is          structure (mapRegion : Region; picture : Bitmap)
type Feature is       structure (featureType : string; featureArea : Area)
type WeatherMap is    structure (satellitePic : Bitmap;
                                 features : set [Feature])
```

Figure 9. Dynamically Generated Representation of New Types

Figure 10 shows part of the system code which processes the source representation and updates the entry points. It uses the system procedure *compileTypes* to compile the source representation into a table which is then used by the procedure *createEntryPoints* to determine the types of the new entry points being created. Note that the code in Figure 9 is specific to this particular example of evolution, while that in Figure 10 is general code used in all occurrences of evolution.

```
let oldEntryPointNames = ! Code to generate list of names
                         ! in this case ["MAPS"]
let newEntryPointNames = ! Code to generate list of names
                         ! in this case ["MAPS","FEATURES","AREAS"]
let newEntryPointTypes = ! Code to generate list of type names
                         ! in this case ["WeatherMap","Feature","Area"]

! compileTypes : proc (SourceRep -> Table [string,TypeRep] )
let typeTable = compileTypes (typeRep) ! typeRep defined in Figure 9

deleteEntryPoints (oldEntryPointNames)
createEntryPoints (newEntryPointNames, newEntryPointTypes, typeTable)
```

Figure 10. Creating and Deleting Entry Points

Next, to maintain consistency the database must create versions of the existing weather map objects with the new type. To allow for the additive change each new object contains an empty set of features; to allow for the descriptive change it contains a *Bitmap* object derived from the original image. The former is catered for automatically by a built-in rule for the **set** constructor, while the user is prompted to supply a conversion function from *image* to *Bitmap*. Figure 11 illustrates the conversion of a weather map object. Note that none of the raw data is actually copied, rather references to the data are manipulated in order to create a new view over it.

Figure 11. Creating a New Version of a Weather Map Object

The creation of the new objects is also achieved using linguistic reflection: the representation of a source code fragment is constructed dynamically, based on details of the schema change, and then transformed into executable code and executed. Another requirement is a dynamic binding mechanism to allow objects with new types to be introduced into the database [MCC+95]. Figure 12 shows the hyper-program source code representation generated by the

system, producing a procedure to convert weather map objects to the new schema. The underlined words indicate direct links to the appropriate types and procedures already present in the database (*imageToBitmap* is a procedure supplied by the user).

```
type Old is structure (satellitePic : image)
type New is structure (satellitePic : Bitmap; features : set [Feature])
proc (oldVersion : Old -> New)
     New (imageToBitmap (oldVersion (satellitePic)),
          emptySet [Feature]())
```

Figure 12. Automatically Generated Representation of Conversion Procedure

Once the hyper-program representation has been generated it is compiled, projected onto the expected type `proc (Old -> New)` with a dynamic type check, and the resulting procedure used to convert the existing weather map objects. Figure 13 shows part of this process:

```
type Old is …        ! declare appropriate types
type New is …
project compile (programRep) as executable onto
proc (Old -> New) : ! apply executable to existing map objects
default :           ! report an error in the generation process
```

Figure 13. Compiling and Applying Conversion Procedure

Some strategies for object conversion with more complex schema changes, in particular descriptive changes, are outlined in [CCK+94a].

The final step is to make the applications which depend on the changed part of the schema (in this case all of them) consistent with the schema change. The *findMap* application can simply be recompiled under the new schema, since its operation does not depend on the structure of *WeatherMap*. The *extractFeatures* application, which is itself performing the schema change, does depend on the details of *WeatherMap* since it needs to access the photographic image which has now become part of the *Bitmap* object. The application hyper-program source code is located and changed appropriately. This edit can be performed entirely by the user, or the system may be able to partly automate the change, requiring the user only to supply a procedure to convert from *Bitmap* back to *image*. Once the updated version of the application is compiled it is substituted for the existing version and the schema update is complete. If it is required that the new version of the *extractFeatures* application continue after the update, it is possible for the old version to record details of its current state in the database before it replaces itself with the new version. These can then be read by the new version which continues execution from the same point. Alternatively these details can be coded into the new version during the process of constructing the source code of the new version.

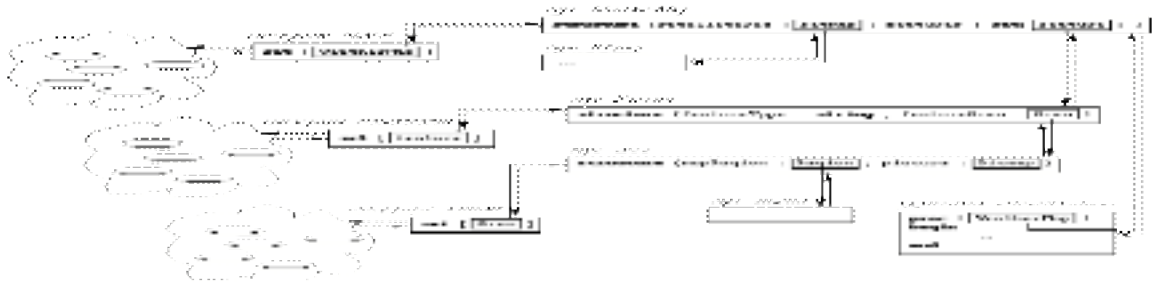

Figure 14. Database After Dynamic Schema Refinement

Figure 14 shows the state of the persistent store after the dynamic schema change, with the bi-directional links in place to support further evolution:

6 Conclusions

Databases that fail to evolve will atrophy and eventually die. Thus for truly long term data, the database must support mechanisms that will allow it to evolve to meet the ever changing needs of applications builders and users. While the problem of evolution has been well recognised, new challenges have been presented to the database community by new classes of application that are typified by their need to store large amounts of information whose structure may change dynamically throughout its lifetime. This requires complex mappings of the data to the schema. The complexity of this task is illustrated by the growth in popularity of specialist databases that allow for evolution in a restricted domain. Here, we postulate that given the correct mechanisms, general purpose databases may accommodate evolution in the desired manner.

Our approach is twofold. We utilise a persistent environment where the data, programs and meta-data may be manipulated in a consistent manner. Programs may access the data, other programs and representations of the meta-data. Using linguistic reflection the schema may be altered to record any new structure discovered by applications in the data, programs or meta-data. Our second approach is to construct or initially store the data in a form that accommodates evolution. All constructions over this data and any derived data are formed by using views. These views may be overlapping, non-contiguous and contain embedded objects. Encapsulation has been abandoned by providing viewing mechanisms that are either open or abstract. Abstract views restrict access to data thus providing the essential needs of information hiding. By constructing the abstract views in the correct manner, even the abstraction may be broken in a controlled manner by the use of trusted passwords.

Some of the techniques in this paper are known. However their combination is unique as is their application to this new set of challenges. We expect such challenges to grow with the work on querying the file [ACM93] which imposes structure dynamically on unstructured data, programming with unstructured data [BDS95], heterogeneity and in attempts to program the WWW, as in Java [GM95] and Internet programming [Con96], which require structure in distributed applications to be discovered and checked locally.

7 Acknowledgements

This work was supported in St Andrews by EPSRC Grant GR/J67611 "Delivering the Benefits of Persistence". Richard Connor is supported by EPSRC Advanced Fellowship B/94/AF/1921.

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. URL: *http://www.dcs.st-and.ac.uk/research/publications/ABC+83a.php*

[ACM93] Abiteboul, S., Cluet, S. & Milo, T. “Querying and Updating the File”. In Proc. 19th International Conference on Very Large Data Bases, Dublin, Ireland, Agrawal, R., Baker, S. & Bell, D. (eds) (1993) pp 73-84, Technical Report ESPRIT BRA Project 6309 FIDE$_2$ FIDE/93/60.

[AGO94] Albano, A., Ghelli, G. & Orsini, R. “Fibonacci Reference Manual: A Preliminary Version”. ESPRIT BRA Project 6309 FIDE$_2$ Technical Report FIDE/94/102 (1994).

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

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

[BDS95] Buneman, P., Davidson, S.B. & Suciu, D. “Programming Constructs for Unstructured Data”. In Proc. 5th International Workshop on Database Programming Languages, Gubbio, Italy, Atzeni, P. & Tannen, V. (eds) (1995) pp 266-276.

[Car84] Cardelli, L. “A Semantics of Multiple Inheritance”. In **Lecture Notes in Computer Science 173**, Kahn, G., MacQueen, D.B. & Plotkin, G. (eds), Springer-Verlag, Proc. International Symposium on the Semantics of Data Types, Sophia-Antipolis, France, Goos, G. & Hartmanis, J. (series ed) (1984) pp 51-67.

[CBM95] Connor, R.C.H., Balasubramaniam, D. & Morrison, R. “Investigating Extension Polymorphism”. In Proc. 5th International Workshop on Database Programming Languages, Gubbio, Italy, Atzeni, P. & Tannen, V. (eds) (1995) pp 13-22.

[CCK+94a] Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Morrison, R. “Using Persistence Technology to Control Schema Evolution”. In Proc. 9th ACM Symposium on Applied Computing, Phoenix, Arizona, Deaton, E., Oppenheim, D., Urban, J. & Berghel, H. (eds) (1994) pp 441-446, Technical Report ESPRIT BRA Project 6309 FIDE2 FIDE/94/97.

[CDM+90] 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. (eds), Springer-Verlag, Proc. 2nd International Conference on Extending Database Technology (EDBT'90), Venice, Italy, Goos, G. & Hartmanis, J. (series ed), ISBN 3-540-52291-3 (1990) pp 301-315.

[Con96] Connor, R.C.H. “InterProgramming”. Personal communication (1996).

[GM95] Gosling, J. & McGilton, H. “The Java™ Language Environment: A White Paper”. Sun Microsystems, Inc (1995). URL: *http://java.sun.com/doc/language_environment/*

[KCC+92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In **Persistent Object Systems**, Albano, A. & Morrison, R. (eds), Springer-Verlag, Proc. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed), ISBN 3-540-19800-8 (1992) pp 86-106.

[MBC+94] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “The Napier88 Reference Manual (Release 2.0)”. University of St Andrews Technical Report CS/94/8 (1994).

[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.

[MM93] Matthes, F. & Müßig, S. "The Tycoon Language TL: An Introduction". University of Hamburg Technical Report DBIS 112-93 (1993).

[SMK+93] Stemple, D., Morrison, R., Kirby, G.N.C. & Connor, R.C.H. "Integrating Reflection, Strong Typing and Static Checking". In Proc. 16th Australian Computer Science Conference (ACSC'93), Brisbane, Australia (1993) pp 83-92, Technical Report University of St Andrews Report CS/93/6.

[SSS+92] Stemple, D., Stanton, R.B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G.N.C., Fegaras, L., Cooper, R.L., Connor, R.C.H., Atkinson, M.P. & Alagic, S. "Type-Safe Linguistic Reflection: A Generator Technology". ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/49 (1992).

[Zdo93] Zdonik, S.B. "Incremental Database Systems: Databases from the Ground Up". In Proc. ACM SIGMOD, Washington D.C., USA (1993) pp 408-412.