Language Bindings for Spatio-Temporal Database Programming in Tripod

Tony Griffiths, Norman W. Paton, Alvaro A. A. Fernandes, Seung-Hyun Jeong, Nassima Djafri
{griffitt|norm|a.fernandes|jeongs|djafrin7}@cs.man.ac.uk

Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, United Kingdom

Abstract. While there are many proposals for spatio-temporal data models and query languages, there is a lack of research into application development using spatio-temporal database systems. This paper seeks to redress the balance by exploring how to support database programming for spatio-temporal object databases, with specific reference to the Tripod spatio-temporal OODBMS.

1 Introduction

Spatio-temporal databases provide facilities for capturing and querying changes that have taken place to the spatial and aspatial properties of data over time. Applications for spatio-temporal databases are potentially numerous, and significant work has been undertaken to develop spatio-temporal data types (e.g. (1)), data models (e.g. (2)), storage structures (e.g. (3)) and algorithms (e.g. (4)). However, to date, few proposals have been worked through to yield complete spatio-temporal database systems, and thus work on application development languages and environments tends to lag behind that on models and algorithms.

There are several techniques that can be used to develop spatio-temporal applications, including bespoke development where the database schema and application programs are developed on a per-application basis, or versioning systems, where transaction time is used to create snapshots of the system data. Perhaps the most straightforward approach to the development of a spatio-temporal database system involves defining spatio-temporal data types, and adding these as primitive types to a relational database system using abstract data type (ADT) mechanisms (also known as "cartridges" or "data blades"). Although spatio-temporal ADTs are not yet commercially available, commercial spatial extensions to relational systems from database and geographical information system vendors are in widespread use.

If adopted to support spatio-temporal data, ADTs would make operations for querying and updating spatio-temporal data available from within a query language, but have a number of limitations for wider application development.

For example, impedance mismatches between a programming language and a database system are increased in the presence of database type extensions. It may therefore not be straightforward for an application to manipulate transient spatio-temporal data using the same types as are stored in the database without creating temporary tables, and casting analysis operations over such data as queries over the database. This damages programmer productivity by increasing the extent to which the application developer is conscious of programming using two loosely integrated languages, which encourage different programming paradigms, and may well have performance implications. Although work on open standards for representing geospatial data (e.g. `www.opengis.org`) could lead to the development of consistent representations in multiple language environments, standard representation for spatio-temporal data are certainly some way off.

As a result, there is a need to explore how best to support database programming for spatio-temporal data. This includes issues such as how database types relate to programming language types, how query language operations relate to those provided in a programming language setting, and what impact the use of imperative language constructs may have for declaratively defined spatio-temporal models. This paper discusses such issues in the context of the Tripod (6) spatio-temporal object database system. The Tripod system is an extension of the ODMG object model (7) to support spatial, timestamp and historical data representation. As illustrated in Figure 1, Tripod not only extends the ODMG object model, but also the OQL query language and the programming language bindings. This paper discusses the design of the Tripod programming language bindings, and illustrates them in use in a simple cadastral application. We know of no previous work that explicitly addresses programming language support for spatio-temporal application development.

The remainder of the paper is structured as follows. Section 2 provides an overview of the Tripod system. Details of the Tripod architecture are presented in Section 3, focussing on how to create a database, and on the mapping between the object model's type system and that of the language bindings. Section 4 illustrates how developers can use the language bindings to program spatio-historical database applications. Finally Section 5 draws some conclusions.

2 Tripod Overview

Tripod is a spatio-historical object database system that orthogonally extends the ODMG standard for object databases. By orthogonality we mean that users can model the spatial, or the temporal aspects of modelled types and their properties, or both, or neither. Figure 1 illustrates the relationships between the different components in Tripod. At the core is the ODMG object model, which is extended with primitive spatial and timestamp types.

The ODMG Object Model provides a set of object and literal types – including collection types, (e.g., `Set`, `Bag` and `List`) and atomic types (e.g., `long` and

`string`) – with which a designer can specify their own object types, and thus construct a particular database schema. Each user-defined type has a structure (a collection of attributes and binary relationships with other user-defined types) and a behaviour (a collection of methods whose implementation is specified using the language binding).

Tripod's spatial data types (SDTs) are based on the ROSE (RObust Spatial Extensions) approach described in (5). The ROSE approach defines an algebra over three SDTs, namely `Points`, `Lines` and `Regions`, and an extensive collection of spatial predicates and operations over these types. Every spatial value in the ROSE algebra is set-based, thus facilitating set-at-a-time processing. Roughly speaking, each element of a `Points` value is a pair of coordinates in the underlying geometry, each element of a `Lines` value is a set of connected line segments, and each element in a `Regions` value is a polygon containing a (potentially empty) set of holes.

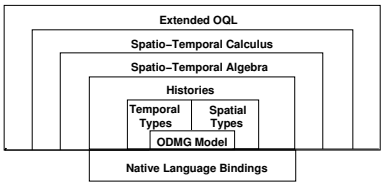


Fig. 1. Tripod Components.

Tripod extends the set of ODMG primitive types with two timestamp types, called `Instants` and `TimeIntervals`. The timestamp types are one-dimensional specialisations of the spatial types, thus inheriting all the functionality of the spatial types, but extending these with timestamp-specific notions such as a calendar and ordering predicates. For further details see (8).

Figure 2 illustrates timestamps in graphical form, where timestamp ***A*** is a `TimeIntervals` value, and timestamps ***B*** and ***C*** are `Instants` values. Notice that ***B*** happens to be a singleton. Tripod's timestamps benefit from the collection-based nature of their spatial counterparts, as they are allowed to contain gaps. This facility is used to good effect in the maintenance of histories as shown in Section 4.

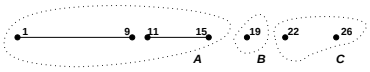


Fig. 2. Example Tripod Timestamps

The Tripod history mechanism provides functionality to support the storage, management and querying of entities that change over time. A history models the changes that an entity (or its attributes, or the relationships it participates in) undergoes as the result of assignments made to it. In the Tripod object model, a request for a history to be maintained can be made for any construct to which a value can be assigned, i.e., a history is a history of changes in value and it records episodes of change by identifying these with a timestamp. Each such value is called a *snapshot*. As a consequence of the possible value assignments that are well defined in the Tripod model, a history can be kept for object identifiers, attribute values, and relationship

instances. In other words, any construct denoted by the left hand side of an assignment operation (i.e., an *l-value*) can have a record kept of the different values assigned to it over time, no matter the type of the value assigned.

3 The Tripod Architecture

Figure 3 details the various components of the Tripod architecture, and in particular how these components interact with each other in the specification of spatio-historical database applications. There are three main components in the Tripod architecture: a Persistent Store that is responsible for loading and saving persistent objects to and from the database and also for maintaining metadata about a particular database schema; a Query Processor that is responsible for optimizing and executing database queries; and a Programming Language Binding that is responsible for providing programming language access to the database, and which is the principal focus of this paper.

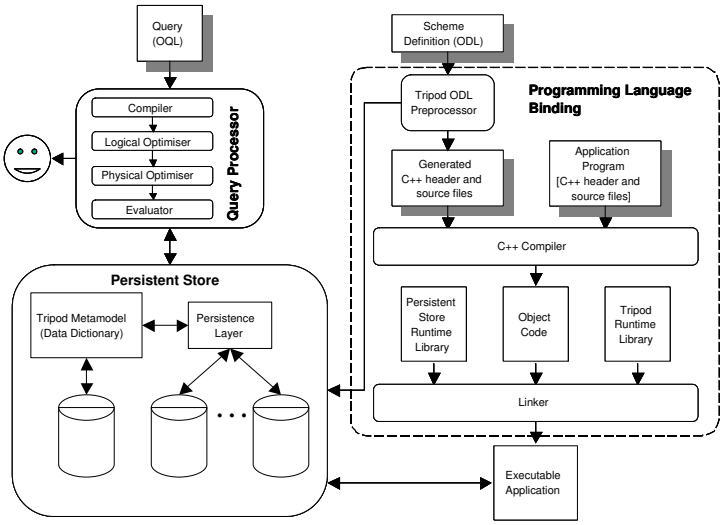


Fig. 3. Tripod Architecture.

3.1 Creating a Tripod Database

Developers specify a Tripod database by defining a schema definition using an extension to ODMG's object definition language (ODL). Figure 4 provides a Tripod ODL specification of a simple cadastre schema that utilises two historical types `Person` and `LandParcel` with a 1:M binary relationship `owns` between them, and a non-historical type `FlagPole`. For each historical property (attribute

or relationship) Tripod maintains a history of the values of that property (primitive values or collections of primitive values for attributes, and object values or collections of object values for relationships) with timestamps (`instants` or `timeIntervals`) at the specified granularity (i.e., `DAY`). When applied to the keyword `class`, the keyword `historical` tells Tripod to maintain an implicit, i.e., system-maintained, attribute called `lifespan`, whose purpose is to record when instances of this class were valid or invalid in a database.

```
forward class Person; forward class LandParcel; forward class FlagPole;

historical<timeIntervals, DAY> class Person ( extent Persons ) {
   historical<timeIntervals, DAY> attribute string name;
   historical<timeIntervals, DAY> relationship set<LandParcel> owns inverse LandParcel::owned_by;
   void purchase(in state<timeIntervals, LandParcel> aParcel);
   void sell(in state<timeIntervals, LandParcel> aParcel); };

historical<timeIntervals, DAY> class LandParcel ( extent Parcels ) {
   attribute string postcode;
   historical<timeIntervals, DAY> attribute regions boundary;
   historical<timeIntervals, DAY> relationship Person owned_by inverse Person::owns; };

class FlagPole ( extent FlagPoles) { attribute points location; };
```

Fig. 4. Example Schema

3.2 Model Mapping

Once a schema has been specified, developers must send the ODL file to Tripod for processing. There are several outputs from this process, namely: An empty Tripod database whose structure corresponds to that of the schema; A separate metadata database (data dictionary) populated with metadata corresponding to the schema (Tripod's metamodel is an orthogonal extension of the ODMG metamodel to accommodate histories); a collection of C++ header files, each of which is a mapping from an ODL class to a corresponding C++ class; and a collection of C++ source files, each of which contains implementations of methods to set and get the class's properties.

Each of these outputs is automatically generated by Tripod from an ODL schema specification. The ODMG standard allows for the functionality provided by these C++ classes to be made available to the language bindings through two mechanisms: the mapped types and embedded queries, both of which are available in Tripod. Figures 5 and 6 show the mapping from the ODL of Figure 4 to the C++ class `LandParcel`. This will be explained in the following sections.

Mapping Classes In conformance with the ODMG standard, Tripod maps *e*ach ODL class to a corresponding C++ class. For each historical class Tripod generates an additional, private and system-maintained, historical attribute called `lifespan` (Figure 5, line 20), whose purpose is to record when objects of this type exist in a database. For example the lifespan: $\{\langle[1-8, 17-27), \texttt{true}\rangle\}$

```
1  #include "Person.h"
2  #include "LetterBox.h"
3  #include "ODMG.h"
4
5  // forward declarations
6  class Person;
7  class LetterBox;
8
9  extern const char __Person_owns[];
10
11 class LandParcel : public Object {
12 // Automatically generated constructor stubs.
13 public: // These must be written by the programmer in a corresponding .cpp file
14 LandParcel();
15 ~LandParcel();
16
17 public: // Automatically generated initialiser stubs
18 LandParcel(const LandParcel&);
19
20 d_History<d_TimeIntervals, DAY,  d_Boolean> __lifespan; // lifespan attribute
21
22 private:  // attributes
23 d_String postcode;
24 d_History<d_TimeIntervals, DAY, d_Regions> boundary;
25
26 public: // relationships
27 d_Historical_Rel_Set<Person, __Person_owns, LandParcel, d_TimeIntervals, DAY> owned_by;
28
29 public:  // operations
30 void merge_with(d_Ref<State<d_TimeIntervals, LandParcel> > aLandParcel);
31 void sell_to(d_Ref<State<d_TimeIntervals, Person> > aPerson);
32
33 // Access methods omitted
34 public: //Extent
35 static d_Ref<d_Set<d_Ref<LandParcel> > > Parcels; // reference to the class extent
36 static const char* extent_name;  };
```

Fig. 5. ODL to C++ mapping for class `LandParcel`

records that the object existed during two distinct time periods (and by inference did not exist at any other times). As per the ODMG standard, Tripod also generates a collection-based and system-maintained attribute (Figure 5, line 35) to represent the extent of all instances of the class.

Mapping Attributes Non-historical attributes are mapped to a corresponding (private) class property with a type from the ODMG C++ type system: i.e., ODL `boolean` maps to `d_Boolean`, whereas ODL `string` maps to `d_String`. Historical attributes, however, map to a new Tripod C++ template type called `d_History` (i.e., `LandParcel` :: `boundary` (Figure 5, line 24)). This type takes as its template arguments the timestamp type, granularity, and snapshot type of the history, and in turn provides facilities to populate, maintain and query itself. Since histories record states, Tripod provides a further C++ template type `State` which takes as arguments the snapshot and timestamp types, and whose purpose is to assist in the process of inserting new states to a history. In conformance

with the ODMG standard, Tripod only allows access to attributes via *set* and *get* methods, which it automatically generates (as shown in Section 4.2).

Mapping Relationships Tripod maps ODL relationships to template types whose purpose is to allow the population of the binary relationship, and to ensure that the objects participating in its roles are automatically maintained. The ODMG C++ binding specifies the types `d_Rel_Ref`, `d_Rel_Set`, and `d_Rel_List` to represent to-one and to-many (unordered and ordered) relationships. Tripod extends the ODMG type system with the types `d_Historical_Rel_Ref`, `d_Historical_Rel_Set`, and `d_Historical_Rel_List`. These types serve the same general purpose as their non-historical counterparts, but instead of an assignment to such a property causing the previous value to be overwritten, they instead add a new state to their internal data structure (a history).

Although it is possible to create spatio-historical schemas using techniques such as bespoke development, versioning or ADT type extensions (i.e., "cartridges"), each of these techniques have their own difficulties. For example: bespoke development requires developers to do all the work themselves without any database kernel or programming language support; versioning depends on the transaction time of any database commits, with valid time being difficult, or impossible, to support; and while ADT extensions to DBMSs provide the necessary kernel database types, the extended models are less expressive than that of Tripod, and require programmers to write explicit code for database maintenance.

4 Programming in Tripod

It is preferable for a historical DBMS to support DBMS functionality over spatial and historical data both orthogonally and synergistically (9). By synergy we mean that if a user makes use of either only spatial or only historical or both spatial and historical facilities, the system responds positively (e.g., by making available specific syntax, by providing additional behaviour, by seeking optimisation opportunities, etc.). This section presents how the language bindings support synergy.

The primary use of the language bindings is to act as an OML for an object database. Although it is entirely possible to query the state of a database through these language bindings, this is not the preferred method. The language bindings do not have full access to the index structures and optimization strategies utilised by Tripod's query language interface (OQL) (10). The preferred method of querying data via the language bindings is to create a `d_OQL_Query` class, and then use the results of this query within an application program. For example:

```
d_Bag<d_Ref<LandParcel> > someParcels;
d_OQL_Query q1("select l from l in Parcels
                where l.boundry.vt.before($1) and l.boundry.value.overlaps($2)");
q1 << aTimeInterval << aSpatialValue;
```

```
d_oql_execute(q1, someParcels);
d_Ref<LandParcel> aParcel; // iterate over results of q1
d_Iterator<d_Ref<LandParcel> > it = someParcels.create_iterator();
while(it.next(aParcel)) {
  cout << "Parcel: " << aParcel->get_postcode() << endl;
  cout << "Area: " << aParcel->get_boundary().CurrentValue().area() << endl;
}
```

This simple example shows how spatial and historical operations can be utilised both within an embedded query and also within the language bindings (see (6) for further details of Tripod OQL).

4.1 Object Creation

The ODMG language bindings specify a syntax and semantics for the creation of both transient and persistent objects. For example, a new persistent `FlagPole` can be created in the database `bncod04` using the following syntax:

```
d_Ref<FlagPole> aFlagPole = new(bncod04, "FlagPole") FlagPole;
```

Tripod extends this syntax with a further overloading of the C++ `new` operator to allow developers to specify an initial valid time for historical objects. Note that in accordance with our aim of synergy, the non-historical version of the `new` operator is available for historical objects, but the valid time of the object will default to a `timeIntervals` value starting at the current system time until it is changed, as returned by the Tripod function `now2uc`().

```
d_Ref<Person> elizabeth = new(bncod04, "Person",
                              "21/4/1926 - until_changed") Person("Elizabeth");
d_Ref<Person> philip = new(bncod04, "Person", "10/6/1921 - until_changed") Person("Philip");
d_Regions r1("regions{[1:3_4:5,4:5_6:3,6:3_3:1,3:1_1:3]}"); // simple rectangle
d_Ref<LandParcel> buckPalace = new(bncod04, "LandParcel",
                                   "1/1/1761 - until_changed") LandParcel("SW1A 1AA", r1);
```

Each of the above historical objects will therefore have a system-maintained `lifespan` historical attribute whose initial values will be the following histories:

1. `elizabeth`: $\{\langle [21/4/1926 - \mathtt{uc}), \mathtt{true}\rangle\}$
2. `philip`: $\{\langle [10/6/1921 - \mathtt{uc}), \mathtt{true}\rangle\}$
3. `buckPalace`: $\{\langle [1/1/1761 - \mathtt{uc}), \mathtt{true}\rangle\}$

where `true` indicates that the object was valid in the database at that time.

4.2 Object Manipulation

Accessing and Assigning Values to Attributes In addition to generating the structure of the C++ class, Tripod automatically generates methods to access an object's properties. So, for example, a non-historical attribute `location : d_Points` will have two methods: `void set_location(d_Points)`,

and `d_Points get_location()`. The signatures for the access methods for the `LandParcel` class are shown in Figure 6.

Since instances of historical attributes are no longer primitive values (they are histories – essentially collections of states), any attempt to access such attributes through their access methods will result in a (possibly empty) history. For example, the operation `buckPalace->get_boundary()` will result in the complete history of Buckingham Palace's boundary, and
`buckPalace->set_boundary(State<d_TimeIntervals, d_Regions>(t1, r1))`
(where `r1` is a `timeInterval` and `r1` is a `d_Regions` value) will result in the assignment of a new state to the `boundary` history. While this operation is more laborious than simply assigning a primitive value, as would happen in a non-historical setting, the nature of a historical domain means that more precision is required. For historical attributes, Tripod generates an overloaded collection of set and get methods. For example, the attribute `LandParcel :: boundary` has two `set_boundary`, and three `get_boundary` methods as shown in Figure 6.

```
1  // Access methods - postcode access omitted for space reasons
2  void set_boundary( const State<d_TimeIntervals, d_Regions >& ) throw(d_Error);
3  void set_boundary( const d_Regions& ) throw(d_Error);
4
5  const d_History<d_TimeIntervals, DAY, d_Regions > get_boundary( );
6
7  const d_History<d_TimeIntervals, DAY, d_Regions > get_boundary( const d_TimeIntervals&,
8                 d_Boolean (d_TimeIntervals::*pred_func) (const d_TimeIntervals*) const  );
9
10 const d_History<d_TimeIntervals, DAY, d_Regions > get_boundary( const d_TimeIntervals&,
11                d_Boolean (d_TimeIntervals::*pred_func) (const d_TimeIntervals*,
12                                                         Quantifier=TPD_FORALL) const,
13                Quantifier=TPD_FORALL  );
14
15 protected: // lifespan attribute
16 void set_lifespan(const d_History<d_TimeIntervals, DAY,  d_Boolean>& _in);
17 const d_History<d_TimeIntervals, DAY,  d_Boolean>& get_lifespan(void) const;
```

Fig. 6. Attribute Access Methods for class `Person`

The first of the set methods (line 2 of Figure 6) allows a new state to be added to the `boundary` history, whereas the method (line 3) allows a regions value to be added to the history with a defaulted valid time of `now2uc()`. The first of the get methods (line 5 of Figure 6) simply returns a copy of the entire history of an object's `boundary` attribute. The remaining two get methods (lines 7 to 8 and lines 10 to 13) have quite complex signatures, but these belie the simple manner in which they are used. Example of these use are shown below.

```
h1 = buckPalace->get_boundary(t1, &d_TimeIntervals::before);
h2 = buckPalace->get_boundary(t2, &d_TimeIntervals::during, TPD_FORALL);
```

Both of these get methods filter the historical attribute based on the timestamp predicate function supplied as the input parameter. The difference between the two variants lies in the non-convex nature of Tripod's timestamps. Since a non-convex timestamp can be composed of several component convex timestamps

(with gaps between them) it may be that part of the argument timestamp satisfies the predicate, but not all of it. Tripod's get methods therefore accept the optional arguments `TPD_FORALL` and `TPD_EXISTS` to allow developers finer-grained control over the semantics of the get operation.

Accessing and Assigning values to Relationships The ODMG language bindings handle relationships in a different manner to attributes. There are no set and get methods per se, rather the syntax uses direct assignment to get/set relationship values. In addition, since relationships are bi-directional, any assignment to one role in the relationship will automatically result in an assignment to the inverse role. In a non-historical setting this is exemplified as follows:

```
// Assume a now non-historical relationship Person::owns
buckPalace->owned_by = elizabeth; // Assignment - automatically updates elizabeth->owns
// The above is equivalent to: (automatically updates buckPalace->owned_by)
// elizabeth->owns.insert_element(buckPalace);
```

Just as historical attributes are no longer simple-valued, historical relationships are themselves collections of states, with each snapshot being either an object identifier or a collection of object identifiers. Since relationships must automatically maintain the integrity of the inverse role, this raises additional complexities for the DBMS in terms of maintaining the integrity of the underlying histories. These complexities should of course be hidden from the user. The code shown below adds a new state to the 1:M `owned_by` relationship of the object `buckPalace`:

```
// Assignment - automatically updates elizabeth->owns
buckPalace->owned_by = State<timeIntervals, Person> \
                       (timeIntervals("21/5/1952 - until_changed"), elizabeth);
// The above is equivalent to:
// elizabeth->owns.insert_element(State<timeIntervals, LandParcel> \
//                                (timeIntervals("21/5/1952 - until_changed"), buckPalace));
```

Figure 7 illustrates the result of the above operation, showing that the object called `elizabeth` has an `owns` history that currently contains a single state whose snapshot is a collection containing a single object `buckPalace`. `buckPalace` has a relationship called `owned_by`, whose single state is the object `elizabeth`.

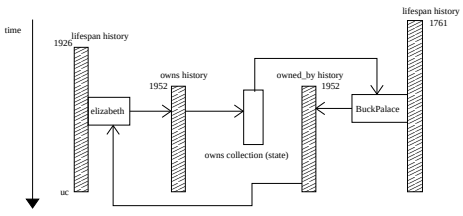


Fig. 7. DBMS Maintenance of Historical Relationships

Manipulating Histories Once a history has been accessed, developers can use the `History` class's API to manipulate the history: i.e., querying, manipulation, performing set-based operations on two histories. Full details of the functionality found in this API can be found in (8). Examples of this functionality in use are:

```
d_History h1 = philip->get_name(); // Retrieve a (copy) of a historical attribute
d_History h2 = elizabeth->get_name();
```

```
// Delete all states before 1/1/1960 (beginning is the earliest data supported by Tripod)
h1.DeleteTimestamp(d_TimeIntervals("beginning - 1/1/1960"));
// Does history contain a particular snapshot value?
d_Boolean b = h2.ContainsSnapshot(State<d_TimeIntervals,d_String>(t1,"Princess Elizabeth"));
d_History h3 = h1 + h2; // Merge two histories: + is overloaded to indicate union
```

In addition to providing functionality to create and populate a history, Tripod provides functionality to query (i.e., `isEmpty`, `Equals`, `ContainsTimestamp`), merge (i.e., `Union`, `Difference`, `Intersection`) and mutate (i.e., `Dissect`) a history (for full details see (8)). Such functionality is available in both the language bindings and the OQL.

Specifying Historical Containment Constraints Tripod's object model enforces temporal containment constraints on the states within a history (i.e., a historical attribute cannot contain states whose timestamps lie outside the scope of the containing object's lifespan (8)). The question therefore arises of what to do if a program attempts to invalidate these constraints. Tripod allows developers to state a policy for such occurrences. They can choose either to: Throw an exception; automatically update the timestamp associated with the new state so that it does not violate the containment constraint; automatically update the lifespan of the containing object so that it now includes the new state's timestamp. Developers can either specify their preference in the Tripod database options file, or can explicitly override these defaults within a program. For example, they can write:

```
TripodPolicy::setHistoricalConstraintPolicy(TripodPolicy::update_lifespan);
// other options are: abort_operation and update_state
```

If an exception is caught then Tripod provides facilities to allow programs to temporarily override the containment constraint policy to, for example, update the containing object's lifespan; alternatively they can update the input state's timestamp or reject the operation.

While the data models that underlie many (spatio-) temporal systems also specify such containment constraints (e.g., (11)), they do not specify what should happen if a violation occurs in any implementation of their model. In a purely temporal setting TSQL2 (12) returns an error code if an attempt is made to inconsistently update temporal tuples, whereas (13) provides greater flexibility by allowing finer-grained spatio-temporal constraints to be specified for moving objects.

4.3 Object Deletion

Semantics of Delete In a non-historical object database, when an object is deleted all references to that object are removed from the database. In the case of a historical database this is not necessarily the case. For example, a user may want to delete an object so that it only exists before or after a certain date, or for

only a certain portion of its valid time. They may however want to permanently delete the object. Each of these types of deletion is a valid operation in terms of a historical database, and as such is supported by Tripod.

Deleting an object for a portion of its valid time affects its system maintained `lifespan` attribute. This, in turn can affect the object's historical attributes, since a object cannot have attributes whose valid time is not contained by the valid time of the object's lifespan.

The consequences of deleting an object may also have ramifications for other objects in a database, through the relationships that the deleted object participated in. For example, if we deleted `buckPalace` from our database, we would want this to be propagated through to its owner, since logically they could not be said to own the land when it did not exist. This type of propagation is also required if only a portion of an object's valid time is deleted, for the same reasons.

The deletion of an object's period of validity in the universe of discourse can be seen to be an operation on its `lifespan` attribute, since it is this (DBMS maintained) property that records when the object exists (and by implication, does not exist) in the database. For example, we could delete `buckPalace` from the database for all valid times after 03/05/2010, if it were demolished on that date. We could also delete a portion of its period of validity if it were destroyed and subsequently rebuilt, i.e., if during the second world war it had been bombed and then rebuilt in 1946. The original valid time of `buckPalace` (as created in Section 4) is realised by its `lifespan` attribute as:

$lifespan_{\texttt{buckPalace}} = \{\langle[1/1/1761 - \texttt{uc}), \texttt{true}\rangle\}$

The first of the above operations (demolishment) would result in a lifespan of:

$lifespan'_{\texttt{buckPalace}} = \{\langle[1/1/1761 - 03/05/2010), \texttt{true}\rangle\}$

It can be seen that this has the effect of updating the object's lifespan from being *quasi*-closed at `until_changed` to being fully closed; whereas the second of the above operations (bombing and rebuilding) results in:

$lifespan''_{\texttt{buckPalace}} = \{\langle[1/1/1761 - 03/04/1942,\ 03/05/1946 - \texttt{uc}), \texttt{true}\rangle\}$

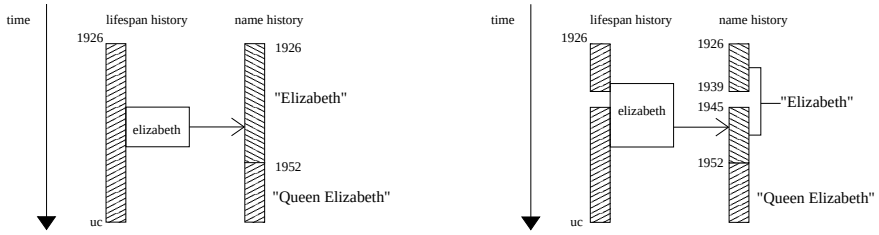
This second operation shows the utility of collection-based timestamps when maintaining Histories. The above operations do not however delete `buckPalace` from the database. If this were the case then `buckPalace.isnull()`[1] would be true.

The question then arises: what if someone attempts to build a visitor centre at `buckPalace` (requiring annexation of land from surrounding areas) on 6/12/2020? Logically, they should not be allowed to undertake this operation since the house no longer exists as it has been demolished. These detailed semantics are dependant on the requirements of the particular application. In this case, the operation should be rejected, whereas another application may reacti-

[1] Note that since Tripod uses a 'smart pointer' to reference persistent objects, then the smart pointer itself will never be NULL, however the object that it references can be NULL, as detected by the `isnull()` function.

vate `buckPalace` so that it merely has a gap in its history (the interpretation of this gap is again application-dependent). Tripod supports both these options.

Effects on an Object's Historical and non-Historical Attributes If an object has a portion of its valid time deleted then this will effect its lifespan property. The affect of any update to an object's lifespan should also be propagated to its attributes. Since Tripod only allows attributes to be simple-valued (i.e., Tripod does not allow object-valued attributes), changes to such attributes need not be propagated to other objects in a database. Figure 8(a) shows the object called `elizabeth` after having it's `name` history populated by two states, as shown in Section 4.2. If this initial system state is updated by deleting a portion of `elizabeth`'s lifespan, then the effect of deleting this portion of `elizabeth`'s lifespan is shown in Figure 8(b), where the corresponding histories have been updated to show the cascading effect of such an update.



(a) Historical `name` Attribute

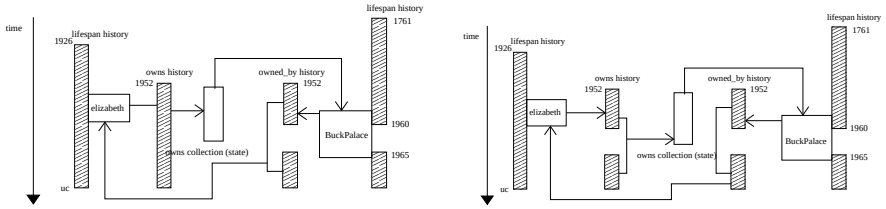
(b) Historical `name` Attribute after Object Deletion

Fig. 8. Updating Historical Attributes

Effects on an Object's Historical and non-Historical Relationships As already stated, if an object has all or portion of its valid time deleted, this can have ramifications for objects that have relationships with the deleted object.

An example of the relationship between two objects has been shown in Figure 7. If we were to delete `buckPalace` during the time period 03/04/1960 to 03/05/1965, then this would have implications for its owners at that time. Figures 9(a) to 9(b) show how the deletion of this portion of `buckPalace`'s lifespan is first propagated to the history that maintains its owners, and then to the inverse of `buckPalace`'s `owned_by` relationship, `Person` :: `owns`.

Language Bindings Extensions to Support Deletion The ODMG 3.0 standard states that an object is deleted by calling the `delete_object`() method on a 'smart pointer' (a `d_Ref`) to that object, i.e.:



(a) Cascading to `buckPalace`'s `owned_by` history

(b) Cascading to related `Person`'s `owns` history

Fig. 9. Updating Historical Relationships

```
d_Ref<FlagPole> fp = new(bncod04, "FlagPole") FlagPole();
fp.delete_object();
```

will cause the Flag Pole object `fp` to be deleted from the database.

Wherever possible, Tripod developers should be able to use the same methods over historical and non-historical objects. Therefore the `delete_object` method is available on both historical and non-historical versions. In the particular case of historical objects, the `delete_object` operation can have several interesting overloadings, taking in several possible semantics, including (amongst many): Delete the object so that it does not exist *during* a specifie timestamp; delete the object so that it does do not exist *before* a specified timestamp; delete the object so that it does not exist *after* a specified timestamp.

These operations can be combined into the following generalised form:

$\texttt{d\_Ref}\langle\texttt{T}\rangle :: \texttt{delete\_object}(\tau, \omega)$;

where `T` is the object type, ω is a Tripod timestamp predicate operation, and τ is a timestamp used in conjunction with ω. For example, this can be instantiated on the object `elizabeth` as:

```
elizabeth.delete_object(d_TimeIntervals("03/04/1960 - 03/05/1965"), &d_TimeIntervals::equals);
elizabeth.delete_object(d_TimeIntervals("03/04/1960 - 03/05/1965"), &d_TimeIntervals::after);
```

Each of these operations will map down to calls on the histories that are being deleted. For a historical object `obj`, this will take the form:

$$\texttt{obj.delete\_object}(\tau, \omega) \equiv \texttt{obj} -> \texttt{lifespan.DeleteTimestamp}(\tau, \omega)$$

In addition, the `delete_object` operation ensures that the lifespans of each of the object's historical attributes and relationships, and the lifespans of the inverses of these relationships are updated so that they satisfy Tripod's historical containment constraints.

Since each class has a different collection of properties, the `delete_object` operation is implemented by extending the ODMG `d_Ref` type with an operation:

```
template<class T, class TS> virtual void d_Ref<T>::delete_object(const TS* timestamp,
                                    d_Boolean(TS::*predicate_func)(const TS*) const) {
   pointed_to_object->delete_object(timestamp, predicate_function);
}
```

that invokes a class-specific `delete_object` operation that Tripod automatically generates from each class's metadata.

Previous database research has identified the delete operator as being of special importance in a temporal setting. However, to the best of our knowledge, such research has been limited to a relational setting rather than an object one. Here, temporal extensions to the relational algebra's delete operator have been proposed (e.g., (14)), allowing access to delete functionality through an extended SQL `DELETE` operation. Examples of such functionality can be found in the TSQL2 language specification (12), and also in (15) where the authors recognise the semantic problems associated with updating (spatio-) temporal data and propose algebraic operators to enforce the consistency of a database. In addition, (16) have developed a temporal RDBMS that takes as input a TSQL2 schema and automatically generates constraints to enforce the temporal consistency of TSQL2 update operations – it is not surprising to note that these constraints are complex in nature.

4.4 Iterating over Historical Properties

Since a history is a collection of states, it exhibits many of the general features of other collection types. One of the important facilities specified by the ODMG language bindings is the ability to iterate over its collection types in ways that are natural to these types. There are two essential properties of such an iteration, namely the type of each element in the collection, and the order in which each element is returned during the iteration.

There are several properties that all ODMG iterators over collection types exhibit as embodied by the base class `Iterator`. Tripod extends these facilities with a new iterator class that provides facilities targeted at a history's specific properties, i.e. such as to: Read or write from or to a History; access a state; move to the next or previous state; move to the Nth state; move forward or backwards N states; test if the iterator is at the beginning or end state.

The adopted approach is to provide an iterator class called `RandomAccessIterator` as described by the interface shown in Figure 10. Tripod's language bindings wrap many of the operations provided by the `RandomAccessIterator` to make them appear more like the interface provided by other iterators found in the C++ standard template library. For example, `goto_nth_position`() is wrapped to appear like the subscript ([]) operator, and `forward_Npositions` is wrapped to appear like the increment (+=) operator.

It is the responsibility of the `d_History` type to specify methods to create an appropriate iterator. The History ADT provides a family of iterator types: `StateIterator`, `SimpleStateIterator` and `InstantStateIterator`, as exemplified by:

```
interface RandomAccessIterator : BidirectionalIterator {
  exception InvalidInde x{ };
  void goto_nth_position(unsigned long index) raises(InvalidIndex);
  void forward_Npositions(unsigned long index) raises(InvalidIndex);
  void reverse_Npositions(unsigned long index) raises(InvalidIndex);
  unsigned long current_position();
};
```

Fig. 10. The `RandomAccessIterator` Interface

$$\texttt{boundary} = \{\langle[t_1 - t_3, t_9 - t_{11}], r_1\rangle, \langle[t_5 - t_7], r_2\rangle, \langle[t_{15} - t_{16}], r_3\rangle\}$$

as follows: `StateIterator` produces elements such as those shown in table 11(a), `SimpleStateIterator` produces elements such as those shown in table 11(b) (i.e., each state has its timestamp decomposed from a non-convex interval to a convex one [2]), and `InstantStateIterator` produces elements as those shown in table 11(c) (i.e., each state is decomposed to a convex instant). An example of the `StateIterator` in use is shown below:

State	value
1	$\langle[t_1 - t_3, t_9 - t_{11}], r_1\rangle$
2	$\langle[t_5 - t_7], r_2\rangle$
3	$\langle[t_{15} - t_{16}], r_3\rangle$

(a) Iterating over $\langle TimeIntervals, snapshot\rangle$ pairs

State	value
1	$\langle[t_1 - t_3], r_1\rangle$
2	$\langle[t_5 - t_7], r_2\rangle$
3	$\langle[t_9 - t_{11}], r_1\rangle$
4	$\langle[t_{15} - t_{16}], r_3\rangle$

(b) Iterating over $\langle TimeInterval, snapshot\rangle$ pairs

State	value
1	$\langle t_1, r_1\rangle$
2	$\langle t_2, r_1\rangle$
3	$\langle t_5, r_2\rangle$
4	$\langle t_6, r_2\rangle$
5	$\langle t_9, r_1\rangle$
6	$\langle t_{10}, r_1\rangle$
7	$\langle t_{15}, r_3\rangle$

(c) Iterating over $\langle Instant, snapshot\rangle$ pairs

Fig. 11. Results of Different Forms of Iteration over a History

```
RandomAccessIterator iter = buckPalace->get_boundary().create_StateIterator(true);
while(!iter.at_end()) {
  try {
    State<d_TimeIntervals, d_Regions> tmp = (State<d_TimeIntervals, d_Regions>)iter.get_element();
    iter.next_position(); // to next state
    iter.previous_position(); // reverse again
    iter.forward_Npositions(3); // forward 3 states
    iter.goto_nth_position(7);  // go directly to state 7
  } catch(const InvalidIndex& error) { std::cerr << error.what() << std::endl; }
}
```

[2] Tripod provides the simple timestamp types `d_Instant` and `d_TimeInterval` for this purpose

5 Summary and Conclusions

OODBMSs provide OML facilities through an imperative programming language interface, and APIs for such interfaces are well documented by standards bodies such as the ODMG. Spatio-historical OODBMSs must provide programming language extensions to accommodate their extended type system and enhanced semantics in ways that facilitate the exploitation of these features by developers migrating from existing OODBMS platforms.

We have previously contended that a historical DBMS should support DBMS functionality over spatial and historical data both orthogonally and synergistically (9). In a relational setting, languages such as TSQL2 reinforce this opinion as in their language features for creating, and maintaining, the temporal aspects of data are optional. In addition, their temporal OML statements frequently provide optional syntax elements assuming temporal default values.

Tripod supports orthogonality by providing an intuitive declarative mechanism that allows developers to create database schemas containing the spatial features of types and properties, or the temporal aspects of modelled types and properties, or both, or neither. This ability provides a high degree of flexibility to the types of systems that can be modelled. In addition, Tripod uses consistent extensions to the existing ODMG 3.0 programming language bindings that allow developers to create, update and delete historical objects and their (spatio-) historical properties. We also provide explicit facilities to ensure that developers can set system policy for determining what should happen if values violate the integrity of the application.

In accordance with our aim of synergy, Tripod provides a specialised syntax to allow the update of (spatio-) historical values. If developers do not use these specialised operations then we provide default semantics for such operations. We have also shown how the complexity of operations such as delete in a temporal setting can be hidden from developers, allowing them to focus on the process of developing their applications.

Although many proposals have been made for temporal and spatio-temporal data models and query languages, we know of no other proposal that follows through from these and also considers close integration of DML functionality in a programming language setting.

Acknowledgments This research is funded by the UK Engineering and Physical Sciences Research Council (EPSRC - grant number GR/L02692) and the Paradigm EPSRC platform grant. Their support is gratefully acknowledged.

Bibliography

[1] Erwig, M., Guting, R., Schneider, M., Vazirigiannis, M.: Abstract and Discrete Modeling of Spatio-Temporal Data Types. Geoinformatica **3** (1999) 269–296

[2] Parent, C., Spaccapietra, S., Zimanyi, E.: Spatio-Temporal Conceptual Models: Data Structures + Space + Time. In: Proc. ACM GIS. (1999) 26–33

[3] Saltenis, S., Jensen, C.S.: Indexing of moving objects for location-based services. In: Proc ICDE02, IEEE Computer Society (2002) 463–472

[4] Lema, J.A.C., Forlizzi, L., Güting, R.H., Nardelli, E., Schneider, M.: Algorithms for moving objects databases. The Computer Journal **46** (2003) 680–712

[5] Güting, R., Schneider, M.: Realm-Based Spatial Data Types: The ROSE Algebra. VLDB Journal **4** (1995) 243–286

[6] Griffiths, T., Fernandes, A.A.A., Paton, N.W., Mason, T., Huang, B., Worboys, M., Johnson, C., Stell, J.: Tripod: A Comprehensive System for the Management of Spatial and Aspatial Historical Objects. In Proc. ACM-GIS, ACM Press (2001) 118–123

[7] Cattell, R.G.G., ed.: The Object Database Standard: ODMG 3.0. Morgan Kaufmann (2000)

[8] Griffiths, T., Fernandes, A., Paton, N., Barr, R.: The Tripod Spatio-Historical Data Model. Data & Knowledge Engineering **49** 1 (2004) 23–65

[9] Paton, N.W., Fernandes, A.A., Griffiths, T.: Spatio-Temporal Databases: Contentions, Components and Consolidation. In: Proc. DEXA 2000 (ASDM 2000), IEEE Press (2000) 851–855

[10] Griffiths, T., A.A.A. Fernandes, Djafri, N., N.W. Paton: A Query Calculus for Spatio-Temporal Object Databases. In: Proc. TIME, IEEE Press (2001) 101–110

[11] Bertino, E., Ferrari, E., Guerrini, G., Merlo, I.: Extending the ODMG Object Model with Time. In: Proc. ECOOP'98. (1998) 41–66

[12] Snodgrass, R.T., ed.: The TSQL2 Temporal Query Language. Kluwer (1995)

[13] Forlizzi, L., Güting, R., Nardelli, E., Schneider, M.: A data model and data structures for moving objects databases. In: ACM SIGMOD Conf. (2000) 319–330

[14] McBrien, P.: Principles of implementing historical databases in RDBMS. In: Proc.BNCOD93, LNCS, Vol. 696, Springer Verlag (1993) 220–237

[15] Lorentzos, N.A., Poulovassilis, A., Small, C.: Manipulation operations for an interval-extended relational model. Data and Knowledge Engineering **17** (1995) 1–29

[16] Detienne, V., Hainaut, J.L.: CASE Tool Support for Temporal Database Design. In Proc. ER 2001. LNCS Vol. 2224, Springer (2001) 208–224