ARTICLE IN PRESS

Available at www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Data & Knowledge Engineering xxx (2003) xxx–xxx

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

The Tripod spatio-historical data model

Tony Griffiths [a,*], Alvaro A.A. Fernandes [a], Norman W. Paton [a], Robert Barr [b]

[a] *Department of Computer Science, University of Manchester, Kilburn Building, Oxford Road, Manchester M13 9PL, UK*
[b] *Department of Geography, University of Manchester, Oxford Road, Manchester M13 9PL, UK*

Received 29 January 2003; received in revised form 7 May 2003; accepted 6 August 2003

Abstract

The storage and analysis of large amounts of time-varying spatial and aspatial data is becoming an important feature of many application domains. This has fuelled the need for spatio-temporal extensions to data models and their associated querying facilities. To date, much of this work has focused on the relational data model, with object data models receiving far less consideration. Where descriptions of such object models do exist, these models fail to fully integrate their spatial, aspatial and temporal dimensions into a uniform and coherent model. In addition, there is currently a lack of systems which build upon these models to produce database architectures that address the broad spectrum of issues related to the delivery of a fully functional spatio-temporal DBMS. This paper presents a foundation for the development of such a system, called *Tripod*, by describing a spatio-historical object model based on a specialized mechanism, called a *history*, for maintaining knowledge about entities that change over time. Key features of the resulting model include: (i) consistent representations of primitive spatial and timestamp types; (ii) a component-based design in which spatial, timestamp and historical extensions are formalized incrementally, for subsequent use together or separately; (iii) compatibility with mainstream query processing frameworks for object databases; and (iv) the integration of the spatio-temporal proposal with the ODMG object database standard. The paper presents a comprehensive formal characterization of the model and illustrates its capabilities in a crime data management application. It is also shown how the model can be programmed using an extension to the ODMG language bindings. The model and language bindings have been fully implemented.

Keywords: Spatial databases; Temporal databases; Spatio-historical object databases; Data modelling

[*] Corresponding author.
E-mail address: tony.griffiths@cs.man.ac.uk (T. Griffiths).

0169-023X/$ - see front matter © 2003 Published by Elsevier B.V.
doi:10.1016/j.datak.2003.08.003

1. Introduction

Spatio-temporal databases have been the focus of considerable research activity in both relational and object settings over the last three decades. Much of the recent work in this area has been surveyed in [13]. However, there are still very few prototypes of complete systems, and far less products that provide effective support for applications tracking changes to spatial and aspatial data over time. We contend that this is because much of the activity in spatio-temporal databases has focused on specific parts of the problem, at the expense of a more holistic view of database systems design and development. It is probably also the case that the database research community has been inclined to undervalue integration or consolidation activities. This section outlines several contentions relating to spatio-temporal databases, with the objective of motivating the need for the data model reported in this paper.

It is our contention that the formal specification of a spatio-temporal object model and the operations that manipulate this model are central to the definition of a complete database system. Moreover, we contend that proposals for spatial and temporal database models and languages often have very large numbers of highly specialized features, which result in difficulties when defining an efficient query processing architecture and object manipulation language (OML).

Another of our contentions is that since spatial and aspatial data change in conceptually similar ways, a single temporal model should be used with both spatial and aspatial data. Previous research has taken two divergent paths towards providing support for spatio-temporal entities. The first provides tightly-coupled mechanisms for spatio-temporal support, whereas the second provides more general support for the temporal aspects of data, of which spatial entities are just one type. The former of these approaches is exemplified by [45], where a spatio-temporal model and algebra are proposed whose entities are spatial objects (simplicial complexes) that have an integral temporal dimension. The second approach is exemplified in a relational setting by [28], where a spatial vector model is proposed in which stored line segments are used as primitives to produce stored polygons. Each of these polygons is then timestamped with its own attribute history using discrete-time semantics. Neither of these techniques have been extended to aspatial data. While such approaches provide special mechanisms for recording the history of purely spatial entities, they fall short of a generalized model for spatial and aspatial data as the some described in this paper. Moreover, neither [45] nor [28] provide any indication of the behaviour of their spatio-temporal entities, i.e., how they are created, updated, queried and destroyed. In contrast this paper describes in detail the bahaviour of spatio-historical entities under each of these circumstances. A more thorough discussion of related work is provided in Section 8.

Database (e.g., [1,34]) and specialist GIS vendors (e.g., [39]) have in recent years introduced products that allow developers to capture the spatial properties of modelled entities. The spatial types presented in this paper provide expressive capabilities that are comparable with the spatial types of these systems. The advance that Tripod brings is in its ability to maintain information about how the spatial and aspatial properties of stored entities change over time in an orthogonal manner. Since this functionality, and that of Tripod's spatial and timestamp types, resides in Tripod's core (kernel) rather than in external or even third-party application programs, we maintain that this holistic approach provides database developers with the essential integrated facilities to store, update, manipulate and query time-varying spatial and aspatial data.

1.1. Tripod: an overview

The Tripod project, from which this paper emerges, is developing a complete spatio-historical database system. The main goal is the extension of the ODMG standard for object databases [7] with facilities for managing vector spatial data, and for the description of past states of both spatial and aspatial data. The key principles underpinning the Tripod project are *orthogonality* and *synergy*. By *orthogonality* is meant that the different extensions to the ODMG standard should be coherent in isolation, so that, for example, the Tripod system should be effective as a historical database in which no spatial data is stored, or as a spatial database in which no use is made of the ability to record historical data. By *synergy* is meant that the system should allow the combined use of spatial and temporal capabilities in a seamless and complementary manner, so that full spatio-temporal applications benefit from integrated facilities without mismatches in the ways different features are supported. Finally, a Tripod database in which neither spatial nor historical features is made use of is an ODMG-compliant database, functionally comparable to those existing in the market (e.g., FastObjects [11]). The Tripod system, which implements all the functionality described in this paper, will be released from `http://www.cs.man.ac.uk/img/tripod/` by the end of September 2003.

Fig. 1 illustrates the relationships between the different components in the design. At the core is the ODMG object model, which the Tripod object model extends. The ODMG model is extended in Tripod with two new categories of primitive types, spatial types and timestamp types. The spatial types are those of the ROSE algebra [23], i.e., the vector types `Points`, `Lines` and `Regions`. The timestamp types are one-dimensional versions of the ROSE algebra types `Points` and `Lines`, referred to as `Instants` and `TimeIntervals`, respectively. The close relationship between the spatial and the timestamp types increases consistency in the representation of the different kinds of data. Past states of objects referring to any ODMG type, including the spatial and timestamp types, can be recorded using a specialized mechanism called a *history*. In essence, a history is a collection of *timestamp-value* pairs, where the timestamp is of a timestamp type and the value is of any of the types in the Tripod extended ODMG model comprising all components in Fig. 1 from Histories inwards.

The layers in Fig. 1 from Histories outwards represent the two interfaces that exist to populate, maintain and query instances of the Tripod spatio-historical object model. Since the ODMG model does not define a declarative OML, developers must use a programming language binding to create, update and delete objects. A more detailed description of the Tripod architecture and

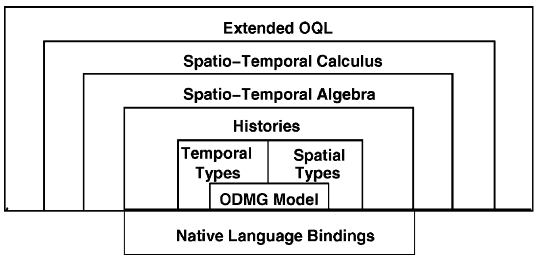

Fig. 1. Tripod's layered architecture.

language bindings can be found in [18]. When the state of a database needs to be queried, developers can either write native language application programs or can issue declarative OQL queries. Tripod's OQL extends that defined by the ODMG with richer possibilities for querying spatial data, timestamp data, and, more importantly, changes in the domain, as reflected in the database. Such queries are mapped to a spatio-historical calculus and then to a spatio-historical algebra. These mappings provide several opportunities for optimization using rewrite rules that are extensions to the techniques used by Fegaras and Maier for optimising object query languages [12]. Examples of these mappings and of the Tripod spatio-historical OQL have been reported in [15]. In addition, [10] gives an indication of the kind of advanced capabilities that the model presented in this paper makes possible.

Both Tripod's native language bindings and its extended OQL utilize the services provided by the spatio-historical object model to access and manipulate spatial and aspatial data. In particular, these interfaces utilize histories whenever changes in data are the primary interest. This paper is structured to reflect Tripod's layered data model, concentrating in particular on the ways in which each layer builds upon the functionality provided by lower levels of the architecture, and how each successive layer conforms to the constraints stemming from these lower layers.

The remainder of this paper is structured as follows. Section 2 presents a case study that is representative of the widespread need to track discrete changes to both spatial and aspatial data over time. Examples from this case study are used throughout the paper. Section 3 describes the ODMG types and the spatial and timestamp literal types with which Tripod extends the ODMG type system. Section 4 describes how Tripod builds on these timestamp types to define a history as an abstract data type (ADT). Section 5 brings together the concepts described in previous sections under the unifying framework of a spatio-historical extension to the ODMG object model. Section 6 presents examples of the Tripod object model in use. Section 7 provides an overview of the implementation of the model within the Tripod system. Section 8 discusses related work, and discussed some of the motivation for the results reported in this paper. Finally, some conclusions are drawn in Section 9 that highlight the contributions of this paper.

2. A motivating example using US census blocks and crime patterns

Following the 1980 census, the US Census Bureau developed the TIGER (Topologically Integrated Geographic Encoding and Referencing) national spatial database [8], whose purpose is to support the identification, storage and manipulation of census block information. Census blocks are the smallest geographical area for which statistics are held, there being some seven million blocks in the US and its associated territories. Census blocks can be aggregated into block groups, which are in turn grouped according to county (or other administrative area).

Fig. 2 shows an example of several blocks that partially define the county called *Baker*, whose indicated boundary completely encloses the blocks 102A, 102B, 314A and 315. A census block is a uniquely numbered closed polygon (often irregular in shape) that is delineated by at least one road, and whose other borders may be one of a number of features, including rail lines, power transmission lines, and water areas (to name but a few). Census blocks are identified such that each block must be wholly contained within a county. Occasionally a block's boundary will need to be adjusted to account for corrections to the geography of its bounding feature. Very occa-

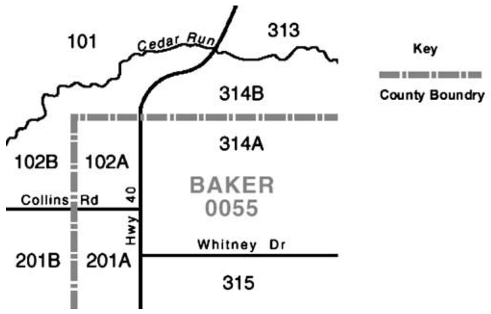

Fig. 2. Example of census blocks within a county.

sionally a block may need to be split/merged if, for example, its population grows to exceed the maximum permitted for a block, or if the update of a block group (such as a County) causes a split. It can be seen that the blocks 102 and 201 in Fig. 2 have been split into blocks 102A, 102B, 201A and 201B to ensure that Baker County contains only non-partitioned blocks.

Blocks can be used by many different agencies to assist them in processing geographically related historical information. For example, city governments need housing information about urban regeneration patterns, property developers need information about how retail space changes over time, and police authorities require detailed statistics about crime patterns to maximize the efficiency and effectiveness in deployment of their resources.

This paper utilizes a hypothetical police crime tracking application as its motivating example. This example is representative of the growing number of applications in this area (for further examples see [4,26,29]) that have been fuelled in the US by the National Institute of Justice's Mapping and Analysis for Public Safety (MAPS) program [33]. The application logs crime incidents (robberies and burglaries) with reference to the building and/or block within which they physically occurred. The application records information that reflects both the time-varying (historical) nature of the crime incident and the physical environment in which the incident occurred. For example, the geographic extent of a building may vary over time as the result of annexation or demolition, and the block within which a building is situated may change over time as the result of block splitting. These processes may be complex operations, especially if the geographic extent contains several holes or even islands within holes. The crime tracking application also needs to ensure that any qualitative constraints between the geographic properties of the various entities are maintained. For example, a building must be enclosed by one and only one block, buildings must not overlap, a block must be bounded by at least one road, to name but a few examples.

The application is also used to relate the location of various suspects, and their alibis, with the information known about crime incidents. The system should also be able to produce aggregate reports at both the block and block group (i.e., county) levels and relate this to block demographic information (i.e., population density).

There are a growing number of crime tracking systems, typified by the Philadelphia project [38] that helps crime professionals maximize their resources. Such systems typically utilize specialist GIS technology such as ArcView [39] to generate crime density maps showing summaries of activity for a user-specified period. In addition, criminologists require computer systems to assist them in building novel applications for the analysis of crime patterns, to assist (for example) in the prediction of crime trends (e.g., [6,44]). The major downside of using such specialist GIS systems is that they only handle spatial data, and must be integrated with mainstream database technology to allow aspatial data to be handled.

Tripod is not a spatial analysis or data mining tool. It does however provide kernel facilities to allow the rapid construction of database-driven systems such as those typified above, providing systems developers with integrated facilities for the management of persistent spatial and aspatial data. Tripod's main advance on existing GIS technology however lies in its ability to record the history, and to provide an efficient and optimized query interface, to such data. This will allow simple crime mapping systems to evolve into the next generation of advanced spatio-historical applications, as typified by our crime tracking system.

A much simplified schema (adapting the notation in [7], with extensions to represent historical data), showing the main classes of interest in our example is presented in Fig. 3. The schema does not specify the operations supported by each class; these are discussed in Section 6. Classes and properties that can change over time are marked with an 'H' symbol (i.e., are historical). The dynamics and inherent constraints of this application are used throughout the remainder of this paper to illustrate the capabilities of the Tripod OM.

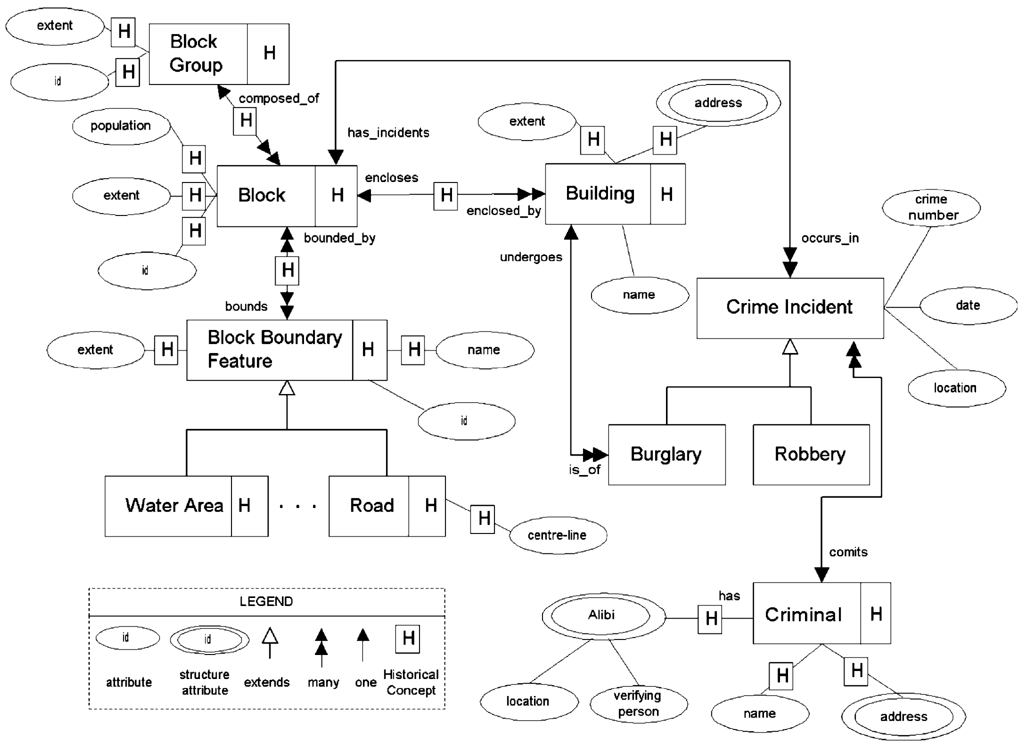

Fig. 3. Simplified crime monitoring schema.

3. The ODMG object model: an overview

This section summarizes the main features of the ODMG object model, concentrating in particular on those features that the Tripod object model extends through its spatio-historical constructs. A more detailed description of the ODMG object model is given in [7].

The basic modelling primitives defined by the ODMG object model are *objects* and *literals*. Objects and literals can be characterized by their *types*, with a particular instantiation of a type being referred to as an *instance*. The model partitions its types into three main categories: atomic types, collection types, and structured types. Atomic object types are user-defined types, (e.g., `Block`, `Burglary`). The Collection object types available are `set`, `bag`, `list`, `array`, and `dictionary`. Instances of collection types are composed of distinct elements of the same type, each of which can be an instance of an atomic type, another collection type, or a literal type. Atomic literal types are exemplified by numeric and character types such as `integer`, `float` and `string`. The model supports several structured literals, i.e., `date` and `time` (although these are superseded by the Tripod timestamp types, for reasons stated in Section 4.1.3). In addition, users can define their own structured literal types through the `struct` type constructor to represent simple record types (e.g., the address of a suspect).

ODMG objects are characterized by their *state* and their *behaviour*, with an object's state being defined by the values of its *properties*. Two kinds of properties are distinguished, namely *attributes* and *relationships*. Attributes are of one object or literal type (e.g., the `Suspect` atomic object type has an attribute called `name` that is of type `string`), whereas relationships are defined between two object types, and may be further categorized as being one-to-one, one-to-many, many-to-one, or many-to-many. Relationships with cardinality greater than one can be implemented by any of the ODMG object model collection types (for example, the relationship called `encloses` is a one-to-many relationship between a `Block` and a collection (set) of `Building` objects). For any relationship, it is possible to traverse the relationship from either of the types participating in the relationship. Thus, for each relationship, two traversal path names must be assigned, one for each direction of the traversal. The `Block` atomic object type therefore declares the collection-valued relationship called `encloses` with the `Building` type, then `Building` must also declare a single valued relationship called `enclosed_by` that is the inverse of the `encloses`.

An object's behaviour is defined by the set of operations that its type defines. These operations are specified as a collection of operation signatures, which define the type of each argument and of any value returned by the operation. For example, the `Block` type may specify a `centroid` operation.

The ODMG object model includes an inheritance mechanism to allow the definition of *generalization–specialization* relationships. Any type that is declared to be a subtype of a particular type may define properties and behaviour in addition to those declared by its supertype. Additionally, the subtype may specialize the existing behaviour or properties of the supertype by overriding the ones it would otherwise have inherited. The collection of all instances of a particular type within a particular database is termed an *extent*. Therefore, if an object is an instance of type `Burglary`, then it is a member of the extent of `Burglary`. Since `Burglary` is a subtype of `Crime_Incident`, then the extent of `Burglary` is a subset of the extent of `Crime_Incident`.

The ODMG Object Model can be used to specify the schema of our crime tracking application using its object definition language (ODL). The following subsections show how Tripod extends the core ODMG type system with new types to support the definition of spatial and timestamp values, and how these extensions can be utilized in an extended ODL.

4. Extensions to the ODMG type system

4.1. Adding spatial and timestamp values to the ODMG type system

4.1.1. The structure of spatial types

Tripod extends the ODMG Object Model literal types with six new structured literal types for representing spatial data. These spatial data types (SDTs) are based on the ROSE (RObust Spatial Extensions) approach described in [23,24]. Underlying the ROSE approach is the notion of a *realm*. A realm is essentially a finite set of points and non-intersecting line segments defined over a discrete grid that forms the ROSE algebra's underlying geometric domain. These properties result in an algebra that has an efficient implementation [21].

4.1.2. The behaviour of spatial types

The ROSE approach defines an algebra over three collection-valued SDTs, namely `Points`, `Lines` and `Regions`, and an extensive collection of spatial predicates and operations (including set operations) over these types. Fig. 5 shows a small subset of the predicate operations; the full set of operations over these types available in Tripod is described in [23]. We use the convention in this paper that all spatial and timestamp operations are `underlined`. Every spatial value in the ROSE algebra is set-based, thus facilitating set-at-a-time processing. Roughly, 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 set of polygons containing a (potentially empty) set of holes.

Some examples of spatial objects that exist within our crime application are shown in Fig. 4. The `Regions` object **R1** represents a school consisting of two disjoint buildings, one containing

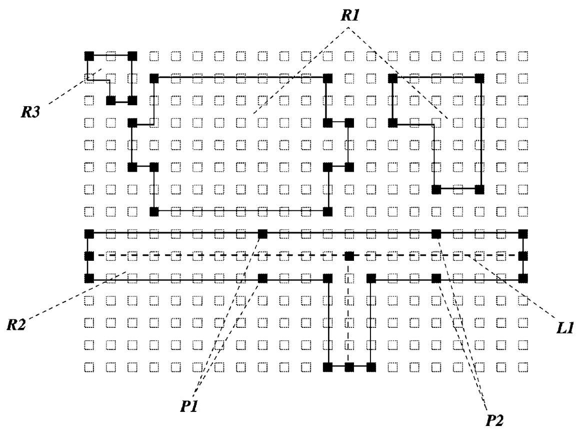

Fig. 4. A 2D-realm and some spatial values defined over it.

an internal quadrangle (i.e., a hole); these two polygonal objects are represented as a single Regions object. Regions object **R3** represents a proposed building to form an annex to the school. Regions object **R2** representing a road whose centre-line is represented by a Lines object **L1**. Objects **P1** and **P2** are Points values representing pairs of road crossing lights (i.e., each value is a pair of lights).

An example of an operation to find the area of the proposed fusion of the Regions values representing the school and its annex (denoted by **R1** and **R3** respectively), and to check that they do not share a common border, can be expressed using the pseudo code in Fig. 6, where plus is the ROSE algebraic operation that computes the union of two spatial values.

To gain access to individual elements within an existing ROSE spatial value, Tripod includes three additional spatial types, namely Point, Line and Region (henceforth referred to as *individual* spatial types). These are conceptually equivalent to the type of an individual element in a ROSE value. While it would be possible to incorporate into the ROSE algebra these three new types with associated sets of operations, Tripod takes a simpler approach by only providing conversion operations from ROSE values into sets of individual spatial values, and vice versa. Therefore, individual spatial types do not support any spatial operations (such as intersection, union, etc.). If such operations are required, then the individual spatial value must first be widened to a ROSE value, as discussed in detail in [23], thus ensuring that the resulting algebra remains closed. Tripod, therefore, extends the ROSE algebra with only two operations, called assemble and disassemble to accommodate the new types. The disassemble operation converts a

adjacent :	Regions	×	Regions	→ bool
area_disjoint :	Regions	×	Regions	→ bool
area_inside :	Regions	×	Regions	→ bool
border_in_common :	Regions	×	Regions	→ bool
common_border :	Regions	×	Lines	→ Lines
encloses :	Regions	×	Regions	→ bool
equal :	Regions	×	Regions	→ bool
intersects :	Regions	×	Lines	→ bool
length :	Lines			→ float
meets :	Regions	×	Lines	→ bool
not_equal :	Regions	×	Regions	→ bool
on_border_of :	Points	×	Lines	→ bool
on_border_of :	Points	×	Regions	→ bool
plus :	Regions	×	Regions	→ Regions

Fig. 5. Example spatial operations.

```
if (not R1.border_in_common(R3)){
  Regions fused_val := R1.plus(R3);
  float fused_area := fused_val.area();
}
```

Fig. 6. Computing the area of fused regions.

<u>assemble</u> : $2^{\text{Point}} \rightarrow$ Points	<u>disassemble</u> : Points $\rightarrow 2^{\text{Point}}$
<u>assemble</u> : $2^{\text{Line}} \rightarrow$ Lines	<u>disassemble</u> : Lines $\rightarrow 2^{\text{Line}}$
<u>assemble</u> : $2^{\text{Region}} \rightarrow$ Regions	<u>disassemble</u> : Regions $\rightarrow 2^{\text{Region}}$

Fig. 7. Conversion from/to collection/individual spatial values.

```
set<Point> Q;
Q := disassemble(P1);
for each p in Q do
    p.x := p.x - 2;
P1 := assemble(Q);
```

Fig. 8. Using `Disassemble` and `Assemble` to operate on individual values.

collection-based ROSE value into a set of the corresponding individual spatial values, whereas <u>assemble</u> does the inverse. These operations have the signatures shown in Fig. 7.

For example, if we wished to model the movement of all elements of the crossing lights P1 two grid points to the left, we could write the pseudo code shown in Fig. 8.

4.1.3. The structure of timestamp types

Tripod extends the set of ODMG primitive types with two timestamp types, called `Instants` and `TimeIntervals`. The underlying domain of interpretation is a structure which we refer to as a *temporal realm* because it is defined to be a one-dimensional specialization of the two-dimensional (spatial) realms defined in [23]. Roughly, a temporal realm is a finite set of integers (whereas a spatial realm is a finite integer grid). Reasons why we adopt this viewpoint and terminology include:

- Realm values are collections, which fit well with the kind of set-a-time strategies that are prevalent in query processing architectures.
- Realm operations are well defined and have a rich set of predicates and constructors with nice closure properties.
- Tripod is a spatio-historical database system and we find it useful (for users and developers) to have realms as a unifying notion for the interpretation of operations on spatial *and* timestamp values.
- This unification at the level of interpretations propagates upwards in the sense that the predicates and operations on realms are defined once and used (possibly after renaming) over both spatial and timestamp values. This also facilitates the reuse of implemented software components, such as those which the authors developed and described in [32].

In a temporal realm, we may think of a time-point as an integer. Then, an `Instants` value is a collection of time-points, and a `TimeIntervals` value is a collection of pairs of time-points where the first element is the start, and the second the end, of a contiguous time-interval. A *timestamp* is either an `Instants` value or a `TimeIntervals` value. Fig. 9 illustrates timestamps in graphical form. In Fig. 9, timestamp ***TI*1** and ***TI*2** are `TimeIntervals` values, and

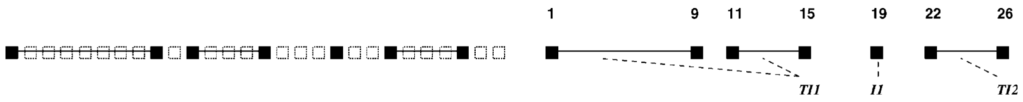

Fig. 9. A 1D-realm and some timestamp values defined over it.

timestamp ***I*1** is an `Instants` value. Notice that ***I*1** and ***TI*2** happen to be singletons. The collection-based nature of the Tripod timestamp types present modellers with new options when developing their applications. For example, a single `Instants` value can be used to record the times when a building was inspected, rather than having to use a set of time values. Also, since a `TimeIntervals` value can have gaps (1D holes), this means that a value from this domain can represent non-contiguous timestamp values, e.g., the period during which a suspect has an alibi.

4.1.4. The behaviour of timestamp types

A temporal realm has an additional property to a spatial realm, viz. a predefined ordering. In the ROSE algebra, there is no predefined notion of one `Points` value being spatially ordered with respect to any other, thus giving application developers the ability to define their own application-specific ordering, within application programs that use the ROSE algebra. Temporal realms, however, must provide some notion of ordering if they are to conform to our intuitions. The Tripod temporal algebra, therefore, extends the ROSE Algebra with ordering predicates based on the underlying order of the temporal realm's integer domain. In addition, the temporal realm utilizes a calendar that maps from the underlying integer domain to one more suited to human cognition. Tripod adopts the standard Gregorian calendar, allowing timestamps to be declared at any of the following collection of granularities: Year, Month, Day, Hour, Minute and Second. Let $\tau \in \{\texttt{Instants}, \texttt{TimeIntervals}\}$. The predicates and operations defined on Tripod timestamps are shown in Figs. 10 and 11. Formal definitions of these operations can be found in [20].

The operation names should suffice to give readers an intuitive understanding of their meaning based on operations on sets of integers (and integer pairs) and on classical definitions for temporal predicates (such as given in [2], and those defined by Ladkin [27] on sets of intervals). The Tripod timestamp predicate operations that are affixed by ω offer alternative and complementary functionality depending on how ω is instantiated. For example, for two temporal values i and j, and a predicate **pred**:

- Do *all* elements in i satisfy the relationship **pred** with some element in j?,

  or
- Does *at least one* element of i satisfy the relationship **pred** with some element in j?

To accommodate these different possibilities, each of the Tripod timestamp predicates has the general form **pred_**$\omega(i, j)$, where $\omega \in \{\forall, \exists\}$. For example, letting $\omega = \forall$ in a template for the **meets** predicate yields the following Tripod predicate **meets_**$\forall$. If we were interested in testing if the `TimeIntervals` values $ti1$ and $ti2$, shown in Fig. 12, to ascertain if they met each other for all their subintervals we would ask: **meets_**$\forall(ti1, ti2)$, or for at least one subinterval we would ask: **meets_**$\exists(ti1, ti2)$.

equals_ω	:	τ	$\times$	τ	$\rightarrow$ bool
nequals_ω	:	τ	$\times$	τ	$\rightarrow$ bool
before_ω	:	τ	$\times$	τ	$\rightarrow$ bool
after_ω	:	τ	$\times$	τ	$\rightarrow$ bool
disjoint_ω	:	τ	$\times$	τ	$\rightarrow$ bool
common_instants_ω	:	τ	$\times$	τ	$\rightarrow$ bool
starts_before	:	τ	$\times$	τ	$\rightarrow$ bool
starts_after	:	τ	$\times$	τ	$\rightarrow$ bool
disjointly_contains	:	τ	$\times$	τ	$\rightarrow$ bool
surrounds	:	τ	$\times$	τ	$\rightarrow$ bool
ends_before	:	τ	$\times$	τ	$\rightarrow$ bool
ends_after	:	τ	$\times$	τ	$\rightarrow$ bool
common_start	:	τ	$\times$	τ	$\rightarrow$ bool
common_end	:	τ	$\times$	τ	$\rightarrow$ bool
contains_ω	:	TimeIntervals	$\times$	Instants	$\rightarrow$ bool
contains_ω	:	TimeIntervals	$\times$	TimeIntervals	$\rightarrow$ bool
meets_ω	:	TimeIntervals	$\times$	TimeIntervals	$\rightarrow$ bool
overlaps_ω	:	TimeIntervals	$\times$	TimeIntervals	$\rightarrow$ bool

Fig. 10. Tripod predicates on timestamp types.

first	:	TimeIntervals			$\rightarrow$ TimeInterval
first	:	Instants			$\rightarrow$ Instant
last	:	TimeIntervals			$\rightarrow$ TimeInterval
last	:	Instants			$\rightarrow$ Instant
vertices	:	TimeIntervals			$\rightarrow$ Instants
lifespan	:	τ			$\rightarrow$ TimeIntervals
no_of_components	:	τ			$\rightarrow$ long
diameter	:	τ			$\rightarrow$ long
duration	:	TimeIntervals			$\rightarrow$ long
distance	:	τ	$\times$	τ	$\rightarrow$ float
intersection	:	τ	$\times$	τ	$\rightarrow \tau$
plus	:	τ	$\times$	τ	$\rightarrow \tau$
minus	:	τ	$\times$	τ	$\rightarrow \tau$

Fig. 11. Tripod operations on timestamp types.

To allow access to individual elements within a Tripod timestamp, two further primitive types, called `Instant` and `TimeInterval` are introduced. These are analogous to the primitive types `Point` and `Line` defined in Section 4.1.1. Consistent with this view, the operations defined for

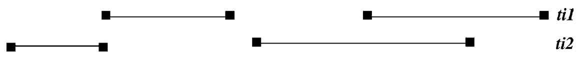

Fig. 12. Comparing `timeIntervals` values.

use with `Instant` and `TimeInterval` are assemble, which maps a set of (`Instant` or `TimeInterval`) values into the corresponding Tripod timestamp, and disassemble, which breaks down a timestamp into a set of (`Instant` or `TimeInterval`) values accordingly. The signatures of these operations follow those shown in Fig. 7 for spatial values.

The discrete domain underlying the temporal realm is bounded by two integer values that correspond to the earliest and latest representable timestamp values. These are given the special names `beginning` and `forever`, and can be referenced as Tripod timestamp literals. Additionally, many applications require the formation of `TimeIntervals` values whose end instant is as yet undecided (i.e., is *pseudo-open* [41]). For example, the timestamp associated with a block's geographic extent could be from 1/1/1990 until some as yet unspecified future date. Tripod uses the special timestamp value `until_changed` to allow such `TimeIntervals` values to be declared. When evaluated during query processing, any value that references `until_changed` is bound to a distinguished `Instants` value representing the current time '*now*', thus having the effect of closing any pseudo-open `TimeIntervals` values. When used in some expressions, `now` can be viewed as a timestamp place-holder that is only instantiated with the current system time when the expression in which it occurs is evaluated.

The complete collection of Tripod literal types can be seen to be a superset of those specified by the ODMG 3.0 standard, with the Tripod timestamp types superseding the ODMG 3.0 timestamp, date and time types. Developers of Tripod schemas can use the new collection of literal types as shown in the examples given in Section 5.

4.2. Adding historical values to the ODMG type system

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.

It should be noted that operations on histories require that all snapshot types provide an equality predicate; this notion is well defined for Tripod literal types, and object identifiers. In the following subsection, a Tripod history is defined as an ADT. The functionality provided by the history ADT is organized into *core* functionality and *secondary* functionality, where secondary functionality can be defined using core functionality.

At first inspection there appear to be a large number of varied and complex operations defined over histories. However, these operations can be divided into a small set of distinct categories that either update, query, merge or mutate history(s). In all cases, the emphasis is on providing database *kernel* functionality that maintain the integrity of a history, and perform well-defined tasks as motivated by the real-world requirements of historical applications. The efficiency of these operations is discussed in Section 4.3 of this paper. These operations are utilized by Tripod's declarative object query language (OQL), and are also available to historical database developers through Tripod's programming language bindings (see Section 7 for further details).

In the following sections, a series of simple scenarios drawn from the domain of our motivating example are used to exemplify the history ADT's operations, where appropriate. A more complete example of histories in use is presented in Section 5.

4.2.1. The structure of histories

A *history* is a quadruple $H = \langle V, \theta, \gamma, \Sigma \rangle$, where V denotes the domain of values whose changes H records, θ is either `Instants` or `TimeIntervals`, γ is the granularity of θ, and Σ is a collection of pairs, called *states*, of the form $\langle \tau, \sigma \rangle$, where τ is a timestamp and σ is a snapshot. In the rest of the paper, let $\mathbb{T}$ denote the set of all timestamps; $\mathbb{V}$, the set of all snapshots; $\mathbb{S}$, the set of all states; and $\mathbb{H}$, the set of all histories.

In a Tripod history, a collection Σ of states is constrained to be an injective function from the set $\mathbb{T}_H$ of all timestamps occurring in H to the set $\mathbb{V}_H$ of all snapshots occurring in H, i.e., for any history H, $S_H : \tau \in \mathbb{T}_H \to \sigma \in \mathbb{V}_H$. Therefore, the following invariants hold, for any history $H = \langle V, \theta, \gamma, \Sigma \rangle$:

(a) Every timestamp occurring in Σ is of type $\theta \in \{$`Instants`, `TimeIntervals`$\}$ and has granularity γ.
(b) For every snapshot σ occurring in Σ, $\sigma \in V$.
(c) A particular timestamp is associated with at most one snapshot, i.e., a history does not record different values as valid at the same time. Note, however, that the snapshot can be an instance of a collection type.
(d) A particular snapshot is associated with at most one timestamp, i.e., all value-equal snapshots within a history are merged to form a single state with a collection-based timestamp, i.e., they are *coalesced*.
(e) For binary operations on histories the granularities are compatible.

The notation used in the rest of the paper is illustrated as follows:

- $[t_i - t_j, \ldots, t_p - t_q)$ represents a `TimeIntervals` value. Each subinterval in this collection value is a *half-open* interval (i.e., the instant representing the lower bound is included, but the one representing the upper bound is not). For reasons of space, simple integer values are used to represent timestamp values rather than calendar-based dates. Although all examples in this paper utilize the `TimeIntervals` type, the `Instants` type could also have been used with appropriate adjustments.
- $\langle [1–5, 6–9), 12 \rangle$ is a single state where 12 is a snapshot value (from the domain of, say, population) that holds from the granule 1 to 5 and from 6 to 9, (i.e., not including instants 5 and 9).

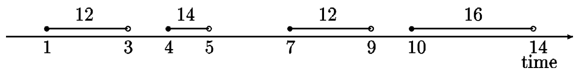

Fig. 13. Example history.

- $\{V, \theta, \gamma, \langle[1\text{–}3, 7\text{–}9), 12\rangle, \langle[4\text{–}5), 14\rangle, \langle[10\text{–}13), 16\rangle\}$ is a history, with $V = \texttt{population}$ and $\theta = \texttt{TimeIntervals}$, graphically shown in Fig. 13, where filled dots indicate a closed bound and a non-filled dot indicates an open bound.

4.2.2. The behaviour of histories

The operations defined by the history ADT can be classified into *constructor*, *update*, *query*, *merge*, and *mutator* operations.

In the following sections, the dot notation is used to denote the individual elements of a particular state. For example, the timestamp of a state s is denoted by $s \cdot \mathsf{t}$, and the corresponding snapshot by $s \cdot \mathsf{v}$. Where reference is made to an operation defined on the underlying timestamp types, the operation name is $\underline{\text{underlined}}$.

Let H (possibly primed or subscripted) range over $\mathbb{H}$. Given H, let S^H denote its state set. Since S^H is a function, we denote its domain, i.e., the timestamps occurring in it, by S^H_T, and we denote its range, i.e., the snapshots occurring in it, by S^H_V. The semantics of an operation ω are sometimes characterized by writing $post(\omega) \Rightarrow \{p_1, \ldots, p_n\}$, where each p_i is a predicate that evaluates to true after ω is carried out. Alternatively, the semantics of ω are sometimes characterized by a rewriting $\omega \equiv (\sigma)[\omega_1, \ldots, \omega_n]$, where each ω_i is a well-defined operation over elements generated in σ.

4.2.2.1. Constructor operations. A history is empty when created, reflecting the situation where an object has a property with, as yet, no recorded values. It is also possible to create a history from the state information contained in another history. The operations have the signatures in Fig. 14 and the following semantics:

Definition 1 (CREATE).

$$post(H := \texttt{Create}()) \Rightarrow \{\texttt{IsEmpty}(H)\}$$
$$H' := \texttt{Create}(H) \equiv$$
$$H' := \texttt{Create}() \uplus (\forall s \in S^H)[\texttt{InsertState}(\texttt{Create}(), \langle s \cdot \mathsf{t}, s \cdot \mathsf{v}\rangle)],$$

where IsEmpty has the obvious semantics, InsertState is defined in Definition 2, and $\uplus$ merges two histories (as defined in Definition 16).

4.2.2.2. Update operations. The update operations provide the ability to insert, delete and update states in a history. These operations preserve the invariant properties of histories. They have the

$$\texttt{Create}: \ \to \mathbb{H}$$
$$\texttt{Create}: \mathbb{H} \to \mathbb{H}$$

Fig. 14. Example history creation operations.

$$\begin{array}{rcl}
\texttt{DeleteTimestamp} & : & \mathbb{H}\times\mathbb{T} \rightarrow \mathbb{H} \\
\texttt{DeleteSnapshot} & : & \mathbb{H}\times\mathbb{V} \rightarrow \mathbb{H} \\
\texttt{DeleteState} & : & \mathbb{H}\times\mathbb{S} \rightarrow \mathbb{H} \\
\texttt{InsertState} & : & \mathbb{H}\times\mathbb{S} \rightarrow \mathbb{H} \\
\texttt{UpdateTimestamp} & : & \mathbb{H}\times\mathbb{S} \rightarrow \mathbb{H} \\
\texttt{UpdateSnapshot} & : & \mathbb{H}\times\mathbb{S} \rightarrow \mathbb{H} \\
\texttt{DeleteAll} & : & \mathbb{H} \qquad \rightarrow \mathbb{H}
\end{array}$$

Fig. 15. Example history update operations.

signatures shown in Fig. 15. Database developers can use these operations through the Tripod native language bindings to populate and maintain properties of objects that have been declared as historical. For reasons of space only a subset of these operations are formally defined here (for full details see [19]). The example shown for update operations represents a series of updates to a suspect's alibi, showing the creation of an alibi, the deletion of a portion of the alibi due to retracted information, and the update of a portion of the alibi due to the verifying person changing their evidence.

The following definitions utilize both the union ($\Cup$) and difference ($\setminus\setminus$) operators over histories. Formal definitions of these operators are given in Definitions 16 and 18. Informally, the $\Cup$ operator returns a new history that contains the states from both its input histories, however, if there are states that share common timestamps then state from the second history takes priority. The $\setminus\setminus$ operator is analogous to the set minus operator in that it returns a history that contains all states of its first history operand that are not present in its second history operand.

Definition 2 (InsertState). The InsertState operation takes a history $H = \langle V, \theta, \gamma, \Sigma \rangle$ and a state $\langle \tau', \sigma' \rangle$, where τ' is of type θ and $\sigma' \in V$, and yields a new history $H' = \langle V, \theta, \gamma, \Sigma' \rangle$. If σ' is equal to some σ occurring in Σ then the timestamp τ associated with σ is recomputed into a timestamp τ_+ that includes τ' (using the $\underline{\texttt{plus}}$ timestamp operator), and $\Sigma' = \Sigma \setminus\setminus_\tau (\{\langle \tau, \sigma \rangle\} \Cup \{\langle \tau_+, \sigma \rangle\})$ If, on the other hand, σ' does not occur in Σ, then Σ is recomputed into a state set Σ_+ that is everywhere equal to Σ except that every state in Σ whose timestamp has common instants with τ' has been recomputed so as to make that no longer the case in Σ_+, and $\Sigma' = \Sigma_+ \Cup \{\langle \tau', \sigma' \rangle\}$. The InsertState operation is a *destructive*, and therefore order-dependent, operation.

$$H' := \texttt{InsertState}(H, \langle \tau', \sigma' \rangle) \equiv$$

$$H' := \texttt{Create}() \Cup \begin{cases} (\exists s \in S^H | s \cdot \mathsf{v} = \sigma')\big[H \setminus\setminus_\tau \big(\{s\} \Cup \{\langle \underline{\texttt{plus}}(\tau', s \cdot \mathsf{t}), \sigma' \rangle\}\big)\big] \\ (\nexists s \in S^H | s \cdot \mathsf{v} = \sigma')\big[\texttt{DeleteTimestamp}(H, \tau') \Cup \{\langle \tau', \sigma' \rangle\}\big] \end{cases}$$

For example, if we want to assert that the alibi for a suspect is that he was located at the coordinate denoted by **P1** during the period [5–8), and that this was verified by their wife, then $\langle \tau', \sigma' \rangle = \langle [5\text{–}8), \mathrm{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle$. If the information contained in the history were: $\Sigma = \{\langle [1\text{–}6), \mathrm{Alibi}(P2, \text{“}\textit{mother}\text{”}) \rangle\}$, then $\Sigma' = \{\langle [1\text{–}5), \mathrm{Alibi}(P2, \text{“}\textit{mother}\text{”}) \rangle, \langle [5\text{–}8), \mathrm{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle\}$, and if $\Sigma = \{\langle [1\text{–}6), \mathrm{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle\}$, then $\Sigma' = \{\langle [1\text{–}8), \mathrm{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle\}$.

Definition 3 (DELETETIMESTAMP). The `DeleteTimestamp` operation takes a history $H = \langle V, \theta, \gamma, \Sigma \rangle$ and a timestamp τ of type θ and yields a new history $H = \langle V, \theta, \gamma, \Sigma' \rangle$. The operation maps Σ into a state set Σ' in which all states in Σ whose timestamp τ' is such that $\underline{\texttt{common\_instants}}(\tau, \tau')$ is true, have been recomputed so that τ does not occur in Σ', otherwise Σ remains unchanged.

$$\begin{aligned}&H' := \texttt{DeleteTimestamp}(H, \tau) \equiv\\ &\quad H' := \texttt{Create}() \uplus (\forall s \in S^H | \underline{\texttt{common\_instants}}(s \cdot \mathsf{t}, \tau))\\ &\qquad [(H \setminus\!\setminus_\tau \{s\}) \uplus \texttt{InsertState(Create ()}, \langle \underline{\texttt{minus}}(s \cdot \mathsf{t}, \tau) s \cdot \mathsf{v} \rangle)]\end{aligned}$$

For example, if $\tau = [2–3)$ and $\Sigma = \{\langle [1–6), \text{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle\}$, then $\Sigma' = \{\langle [1–2, 3–6), \text{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle\}$, reflecting new information that we now know that the suspect does not have an alibi during the time period [2–3).

It can be seen from the definition of the `DeleteTimestamp` operation that if the input timestamp has common instants with the timestamp of any existing state, then this state is initially deleted (using the specialized difference operator ($\setminus\!\setminus_\tau$) on histories, defined in Definition 18 below), and a new state is inserted whose new timestamp is computed using the timestamp $\underline{\texttt{minus}}$ operator, and whose snapshot is the same as in the deleted state.

Definition 4 (DELETESNAPSHOT). The `DeleteSnapshot` operation takes a history $H = \langle V, \theta, \gamma, \Sigma \rangle$ and a snapshot $\sigma \in V$ and deletes the state (if any) within H that matches σ. The operation maps Σ into a state set Σ' in which no state in Σ' contains σ.

$$\begin{aligned}&H' := \texttt{DeleteSnapshot}(H, \sigma) \equiv\\ &\quad H' := \texttt{Create}() \uplus (\forall s \in S^H | s \cdot \mathsf{v} \neq \sigma)\\ &\qquad [\texttt{InsertState(Create()}\langle s \cdot \mathsf{t}, s \cdot \mathsf{v} \rangle)]\end{aligned}$$

For example, if $\sigma = \text{Alibi}(P1, \text{“}\textit{wife}\text{”})$ and $\Sigma = \{\langle [1–6), \text{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle, \langle [9–11), \text{Alibi}(P3, \text{“}\textit{brother}\text{”}) \rangle\}$, then $\Sigma' = \{\langle [9–11), \text{Alibi}(P3, \text{“}\textit{brother}\text{”}) \rangle\}$, reflecting information received that the suspect's wife is an unreliable witness.

Definition 5 (DELETESTATE). The `DeleteState` operation takes a history $H = \langle V, \theta, \gamma, \Sigma \rangle$, a snapshot $\sigma \in V$ and a timestamp τ of type θ and yields a new history $H = \langle V, \theta, \gamma, \Sigma' \rangle$. The operation maps Σ into a state set Σ' in which all states in Σ whose snapshot equals σ and whose timestamp τ' is such that $\underline{\text{common\_instants}}(\tau, \tau')$ is true, are recomputed so that $\langle \tau, \sigma \rangle$ does not occur in Σ', otherwise Σ remains unchanged.

$$\begin{aligned}&H' := \texttt{DeleteState}(H, \langle \tau, \sigma \rangle) \equiv\\ &\quad H' := \texttt{Create}() \uplus (\forall s \in S^H | s \cdot \mathsf{v} = \sigma)\\ &\qquad [(H \setminus\!\setminus_\tau \{s\}) \uplus \texttt{InsertState(Create()}, \langle \underline{\texttt{minus}}(s \cdot \mathsf{t}, \tau), s \cdot \mathsf{v} \rangle)]\end{aligned}$$

For example, if we wished to delete only a portion of the information contained about the location of the suspect validated by their wife, then if $\sigma = \text{Alibi}(P1, \text{“}\textit{wife}\text{”})$, $\tau = [4–6)$ and $\Sigma = \{\langle [1–8, 10–15), \text{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle, \langle [17–21), \text{Alibi}(P3, \text{“}\textit{brother}\text{”}) \rangle\}$, then $\Sigma' = \{\langle [1–4, 6–8, 10–15), \text{Alibi}(P1, \text{“}\textit{wife}\text{”}) \rangle, \langle [17–21), \text{Alibi}(P3, \text{“}\textit{brother}\text{”}) \rangle\}$. However, were $\sigma = \text{Alibi}(P3, \text{“}\textit{brother}\text{”})$ and $\tau = [4–6)$, then Σ would be unchanged.

4.2.2.3. Query operations. This section defines the operations that higher-level layers of the Tripod architecture use to query the values contained within a given history. Such operations can be classified into Boolean and retrieval operations. Their signatures are given in Fig. 16. In addition to the core Boolean and retrieval operations defined in Fig. 16, several secondary operations can be defined in terms of those operations.

Definition 6 (IsEmpty). The `IsEmpty` operation tests whether a history H contains any state.

$$\texttt{IsEmpty}(H) \equiv (S^H = \emptyset)$$

Definition 7 (ContainsTimestamp). The `ContainsTimestamp_`ω operation is in fact a template for a collection of signatures parameterized on any element of the set of predicate operations on Tripod timestamps. A binding for ω specifies the nature of the relationship between the input timestamp and the timestamp of each state within the history operand. For example, given that $\underline{\texttt{before}}$ is a member of that set, letting $\omega = \underline{\texttt{before}}$ in the template yields the following signature $\texttt{ContainsTimestamp\_}\underline{\texttt{before}}$: $\mathbb{H} \times \mathbb{T} \to \texttt{bool}$. The full collection of the available timestamp predicates can be found in 10.

$$\texttt{ContainsTimestamp\_}\omega(H, \tau) \equiv \text{return}(\exists \tau' \in S_T^H)[\omega(\tau, \tau')]$$

For example, if we wanted to see if a suspect had an alibi before a specified time period, and the state sets of two histories H_1 and H_2, both with identical γ, were $\Sigma_1 = \{\langle[1\text{–}6), \text{Alibi}(P1, \text{“}\textit{wife}\text{”})\rangle, \langle[9\text{–}11), \text{Alibi}(P3, \text{“}\textit{brother}\text{”})\rangle\}$ and $\Sigma_2 = \{\langle[5\text{–}10), \text{Alibi}(P2, \text{“}\textit{mother}\text{”})\rangle, \langle[13\text{–}20), \text{Alibi}(P5, \text{“}\textit{friend}\text{”})\rangle\}$ then $\texttt{ContainsTimestamp\_}\underline{\texttt{before}}(H_1, [9\text{–}10))$ would return `true` and $\texttt{ContainsTimestamp\_}\underline{\texttt{after}}(H_2, [21\text{–}22))$ would return `false`.

IsEmpty	:	$\mathbb{H}$	$\to$	**bool**
ContainsTimestamp_ω	:	$\mathbb{H} \times \mathbb{T}$	$\to$	**bool**
ContainsSnapshot	:	$\mathbb{H} \times \mathbb{V}$	$\to$	**bool**
ContainsState_ω	:	$\mathbb{H} \times \mathbb{S}$	$\to$	**bool**
FilterByTimestamp_ω	:	$\mathbb{H} \times \mathbb{T}$	$\to$	$\mathbb{H}$
FilterBySnapshot	:	$\mathbb{H} \times \mathbb{V}$	$\to$	$\mathbb{H}$
FilterByState_ω	:	$\mathbb{H} \times \mathbb{S}$	$\to$	$\mathbb{H}$
EarliestState	:	$\mathbb{H}$	$\to$	$\mathbb{S}$
LatestState	:	$\mathbb{H}$	$\to$	$\mathbb{S}$
CurrentValue	:	$\mathbb{H}$	$\to$	$\mathbb{S}$
GetLifespan	:	$\mathbb{H}$	$\to$	$\mathbb{T}$
GetMaximalLifespan	:	$\mathbb{H}$	$\to$	$\mathbb{T}$
SubsetOf	:	$\mathbb{H} \times \mathbb{H}$	$\to$	**bool**
StrictSubsetOf	:	$\mathbb{H} \times \mathbb{H}$	$\to$	**bool**
Equals	:	$\mathbb{H} \times \mathbb{H}$	$\to$	**bool**
Brackets	:	$\mathbb{H} \times \mathbb{H}$	$\to$	**bool**

Fig. 16. Example history query operations.

Definition 8 (ContainsSnapshot). The ContainsSnapshot operation tests whether a snapshot σ' is contained within a history $H = \langle V, \theta, \gamma, \Sigma\rangle$ with $\sigma' \in V$.

$$\texttt{ContainsSnapshot}(H, \sigma') \equiv (\exists \sigma \in S_V^H)[\sigma = \sigma']$$

For example, if the state set of a history H with $V = \texttt{int}$, $\theta = \texttt{TimeIntervals}$ is $\Sigma = \{\langle[1–6), \text{Alibi}(P1, \text{“}\textit{wife}\text{”})\rangle, \langle[9–11), \text{Alibi}(P2, \text{“}\textit{mother}\text{”})\rangle\}$ then $\texttt{ContainsSnapshot}(H,$ Alibi(P1, “wife”)) would return true and $\texttt{ContainsSnapshot}(H,$ Alibi(P7, “sister”)) would return false.

Definition 9 (FilterByTimestamp). The FilterByTimestamp_ω operation takes as arguments a history $H = \langle V, \theta, \gamma, \Sigma\rangle$ and a timestamp value τ where τ is of type θ, and returns another history $H' = \langle V, \theta, \gamma, \Sigma'\rangle$, containing just those states whose timestamps have the relationship ω with τ.

$$\begin{aligned}&H' := \texttt{FilterByTimestamp\_}\omega(H, \tau) \equiv\\ &\quad H' := \texttt{Create()} \uplus (\forall s \in S^H | \underline{\omega}(\tau, s \cdot \mathsf{t}))\\ &\qquad [\texttt{InsertState}(\texttt{Create()}, \langle s \cdot \mathsf{t}, s \cdot \mathsf{v}\rangle)]\end{aligned}$$

For example, if the state set of a history H is $\Sigma = \{\langle[7–9), \text{Alibi}(P1, \text{“}\textit{wife}\text{”})\rangle, \langle[1–3, 11–15), \text{Alibi}(P3, \text{“}\textit{brother}\text{”})\rangle\}$, then $\texttt{FilterByTimestamp\_}\underline{\texttt{starts\_before}}(H, [6–11))$ produces H' whose state set $\Sigma' = \{\langle[1–3, 11–15), \text{Alibi}(P3, \text{“}\textit{brother}\text{”})\rangle\}$.

Definition 10 (FilterBySnapshot). The FilterBySnapshot operation takes as arguments a history $H = \langle V, \theta, \gamma, \Sigma\rangle$ and a snapshot value σ where $\sigma \in V$, and returns another history $H' = \langle V, \theta, \gamma, \Sigma'\rangle$, whose state set Σ' is a subset of Σ, such that $dom(S_V^Q) = \{\sigma\}$.

$$\begin{aligned}&H' := \texttt{FilterBySnapshot}(H, \sigma) \equiv\\ &\quad H' := \texttt{Create()} \uplus (\forall s \in S^H | s \cdot \mathsf{v} = \sigma)\\ &\qquad [\texttt{InsertState}(\texttt{Create()}, \langle s \cdot \mathsf{t}, \mathsf{v}\rangle)]\end{aligned}$$

Note that FilterBySnapshot always returns a singleton or an empty set.

For example, if the state set of a history H is $\Sigma = \{\langle[1–6), \text{Alibi}(P1, \text{“}\textit{wife}\text{”})\rangle, \langle[9–11), \text{Alibi}(P3, \text{“}\textit{brother}\text{”})\rangle\}$, then $\texttt{FilterBySnapshot}(H, \text{Alibi}(P1, \text{“}\textit{wife}\text{”}))$ produces a history H' whose state set $\Sigma' = \{\langle[1–6), \text{Alibi}(P1, \text{“}\textit{wife}\text{”})\rangle\}$, whereas FilterBySnapshot $(H, \text{Alibi}(P5, \text{“}\textit{sister}\text{”}))$ produces a history with an empty state set.

Definition 11 (EarliestState). The EarliestState operation takes as arguments a history $H = \langle V, \theta, \gamma, \Sigma\rangle$ and returns the state, $\langle\tau, \sigma\rangle$, that has the earliest timestamp in H. If Σ is empty then an exception is raised.

$$\begin{aligned}&\langle\tau, \sigma\rangle := \texttt{EarliestState}(H) \equiv\\ &\quad (\exists s_1 \in S^H, \forall s_2 \in S^H, s_1 \neq s_2, S^H \neq \emptyset)\\ &\qquad [\underline{\texttt{starts\_before}}(s_1 \cdot \mathsf{t}, s_2 \cdot \mathsf{t})]\langle\tau, \sigma\rangle := s_1\end{aligned}$$

For example, if the state set of a history H, representing the population of a block, is $\Sigma = \{\langle[1\text{–}6), 12\,000\rangle, \langle[6\text{–}11), 12\,800\rangle, \langle[11\text{–}18), 13\,100\rangle\}$ then `EarliestState` $(H) = \langle[1\text{–}6), 12\,000\rangle$.

Definition 12 (GetLifespan). The `GetLifespan` operation returns a timestamp value which coalesces the timestamps of all the states in a history. If the history is empty then the empty timestamp is returned.

$$\tau' := \texttt{GetLifespan}(H) \equiv \tau' := (\forall \tau \in S_T^H)[\underline{\texttt{plus}}(\tau, \tau')]$$

For example, if the state set of a history H, representing the known aliases of a suspect is $\Sigma = \{\langle[1\text{–}6, 23\text{–}34), \text{“}Joe\text{”}\rangle, \langle[9\text{–}11), \text{“}Fred\text{”}\rangle\}$ then `GetLifespan` $(H) = [1\text{–}6, 9\text{–}11, 23\text{–}34)$ denotes the periods for which the criminal is known to have used a known alias.

Several predicate operations are also available on histories. For example, one can test if one history is a subset or a strict subset of another history, or whether one history is equal to another history using the `SubsetOf`, `StrictSubsetOf` and `Equals` predicates. Additionally, the `Brackets` predicate tests if the lifespan of one history is contained within or is equal to the lifespan of another.

Definition 13 (SubsetOf). The `SubsetOf` operation takes two histories H_1 and H_2 as its arguments and returns true if H_1 and H_2 have any states with equal snapshot values whose timestamps share common instants.

$$\texttt{SubsetOf}(H_1, H_2) \equiv (\exists s_1 \in S_1^H, \exists s_2 \in S_2^H | \underline{\texttt{common\_instants}}(s_1 \cdot \mathsf{t}, s_2 \cdot \mathsf{t}) \wedge s_1 \cdot \mathsf{v} = s_2 \cdot \mathsf{v})[\,]$$

For example, if the state sets of two histories H_1 and H_2, representing the population history of two blocks are $\Sigma_1 = \{\langle[1\text{–}6), 12\,000\rangle, \langle[9\text{–}11), 14\,000\rangle\}$ and $\Sigma_2 = \{\langle[5\text{–}10), 12\,000\rangle, \langle[13\text{–}20), 15\,000\rangle\}$ then `SubsetOf`$(H_1, H_2) =$ `true`, i.e., there were instants in which the two blocks had the same population.

Definition 14 (Equals). The `Equals` operation takes two histories H_1 and H_2 as its arguments and returns true if each state of H_1 is equal to each state of H_2. The equality is both of the timestamp and of the snapshot, for each state.

$$\texttt{Equals}(H_1, H_2) \equiv \text{return}(\forall s_1 \in S_1^H, \forall s_2 \in S_2^H | \underline{\texttt{equals}}(s_1 \cdot \mathsf{t}, s_2 \cdot \mathsf{t}) \wedge s_1 \cdot \mathsf{v} = s_2 \cdot \mathsf{v})[\,]$$

For example, if the state sets of two histories H_1 and H_2, representing the alibis of two suspects, are $\Sigma_1 = \{\langle[1\text{–}6), \text{Alibi}(P1, \text{“}Sarah\text{”})\rangle, \langle[9\text{–}11), \text{Alibi}(P4, \text{“}Mary\text{”})\rangle\}$ and $\Sigma_2 = \{\langle[1\text{–}6], \text{Alibi}(P1, \text{“}Sarah\text{”})\rangle\}$ then `Equals`(H_1, H_2) would return `false`, whereas if the second state in H_1 were deleted, then `Equals`(H_1, H_2) would return `true`, showing that both suspects shared the same alibis for the same points in time.

Definition 15 (Brackets). `Brackets` is true of two Histories H_1 and H_2 if the lifespan of H_2 is contained within or is equal to the lifespan of H_1.

Let $H_1^s := \texttt{EarliestState}(H_1) \cdot \texttt{t}$,
$H_1^e := \texttt{LatestState}(H_1) \cdot \texttt{t}$,
$H_2^s := \texttt{EarliestState}(H_2) \cdot \texttt{t}$, and
$H_2^e := \texttt{LatestState}(H_2) \cdot \texttt{t}$, then

$$\begin{aligned}
&\texttt{Brackets}(H_1, H_2) \equiv \\
&\quad (\underline{\texttt{starts\_before}}(H_1^s, H_2^s) \vee \underline{\texttt{equals}}(H_1^s, H_2^s)) \\
&\qquad \wedge (\underline{\texttt{starts\_after}}(H_1^e, H_2^e) \vee \underline{\texttt{equals}}(H_1^e, H_2^e))
\end{aligned}$$

For example, if the state sets of two histories, representing alibis of two suspects, H_1 and H_2 are $\Sigma_1 = \{\langle[1\text{–}6), \text{Alibi}(P1, \text{“}\textit{Sarah}\text{”})\rangle, \langle[9\text{–}11), \text{Alibi}(P4, \text{“}\textit{Mary}\text{”})\rangle\}$ and $\Sigma_2 = \{\langle[2\text{–}11), \text{Alibi}(P1, \text{“}\textit{Sarah}\text{”})\rangle\}$ then $\texttt{Brackets}(H_1, H_2)$ would return `true`. The `Brackets` operation is also used in the Tripod object model to state various constraints on object states, as described in Section 5.5.

4.2.2.4. Merge operations. Tripod provides five binary operations that take two histories and, from them, characterize a third (Fig. 17).

Definition 16 (Union). The `Union` operation, denoted by $\Cup$, takes two histories $H_1 = \langle V, \theta, \gamma, \Sigma_1\rangle$ and $H_2 = \langle V, \theta, \gamma, \Sigma_2\rangle$ as its arguments and returns a history $H_3 = \langle V, \theta, \gamma, \Sigma_3\rangle$, where Σ_3 is the set union of Σ_1 and Σ_2, except that the state in the second argument is chosen whenever there is a state in the first argument that shares instants with the second but has a different snapshot. This is to satisfy the invariants that characterize histories.

$$H_3 := \Cup(H_1, H_2) \equiv H_3 := \texttt{Create}(H_1)\ (\forall s_2 \in S^{H_2})[\texttt{InsertState}(H_3, \langle s_2 \cdot \texttt{t}, s_2 \cdot \texttt{v}\rangle)]$$

For example, if the state sets of two histories H_1 and H_2, representing the population history of two blocks are $\Sigma_1 = \{\langle[1\text{–}6), 12\,000\rangle, \langle[9\text{–}11), 14\,000\rangle\}$ and $\Sigma_2 = \{\langle[5\text{–}10), 13\,100\rangle, \langle[13\text{–}20), 15\,000\rangle\}$ then the state set of $H_3 = H_1 \Cup H_2$ is $\Sigma_3 = \{\langle[1\text{–}5), 12\,000\rangle, \langle[5\text{–}10), 13\,100\rangle, \langle[10\text{–}11), 14\,000\rangle, \langle[13\text{–}20), 15\,000\rangle\}$.

The semantics of this operation may not be what the reader expects, as states from the second history input parameter take precedence over those of the first, resulting in a history that does not contain all states from both input histories. The alternative however is to preserve the snapshots from both input histories by changing the snapshot type of the resulting history to a structure that

$$\begin{aligned}
\Cup &: \mathbb{H} \times \mathbb{H} \to \mathbb{H} \\
\Cap_{\mathcal{T}} &: \mathbb{H} \times \mathbb{H} \to \mathbb{H} \\
\Cap_{\sigma} &: \mathbb{H} \times \mathbb{H} \to \mathbb{H} \\
\backslash\!\backslash_{\mathcal{T}} &: \mathbb{H} \times \mathbb{H} \to \mathbb{H} \\
\backslash\!\backslash_{\sigma} &: \mathbb{H} \times \mathbb{H} \to \mathbb{H}
\end{aligned}$$

Fig. 17. Example history merge operations.

is the amalgam of both input types. This however would invalidate the closure properties of our algebra and is therefore avoided.

Definition 17 (Intersection). The intersection operation constructs a new history H_3 from two histories H_1 and H_2. Its result contains the states that are members of both H_1 and H_2. This operation has two variants: one, denoted by $\Cap_\tau$, tests for shared instants in the timestamp values; the other, denoted by $\Cap_\sigma$ tests for equality of snapshot values. Once again any state from H_2 that is value-equivalent to a state from H_1 takes precedence over the latter in the result.

$$\begin{aligned}
&H_3 := \Cap_\tau(H_1, H_2) \equiv \\
&\quad H_3 := \texttt{Create()}\ (\forall s_1 \in S^{H_1}, \forall s_2 \in S^{H_2} | \underline{\texttt{common\_instants}}(s_1 \cdot \mathsf{v}, s_2 \cdot \mathsf{v})) \\
&\qquad [\texttt{InsertState}(H_3, \langle \underline{\texttt{intersection}}(s_1 \cdot \mathsf{t}, s_2 \cdot \mathsf{t}), s_2 \cdot \mathsf{v}\rangle)]
\end{aligned}$$

$$\begin{aligned}
&H_3 := \Cap_\sigma(H_1, H_2) \equiv \\
&\quad H_3 := \texttt{Create()} \bullet (\forall s_1 \in S^{H_1}, \forall s_2 \in S^{H_2} | s_1 \cdot \mathsf{v} = s_2 \cdot \mathsf{v}) \\
&\qquad [\texttt{InsertState}(H_3, \langle \underline{\texttt{intersection}}(s_1 \cdot \mathsf{t}, s_2 \cdot \mathsf{t}), s_2 \cdot \mathsf{v}\rangle)]
\end{aligned}$$

For example, if the state sets of two histories H_1 and H_2, representing the alibis of two suspects, are $\Sigma_1 = \{\langle [1\text{–}6), \text{Alibi}(P1, \text{“}\textit{Sarah}\text{”})\rangle, \langle [9\text{–}11), \text{Alibi}(P4, \text{“}\textit{Mary}\text{”})\rangle\}$ and $\Sigma_2 = \{\langle [5\text{–}10), \text{Alibi}(P2, \text{“}\textit{Joan}\text{”})\rangle, \langle [13\text{–}20), \text{Alibi}(P1, \text{“}\textit{Sarah}\text{”})\rangle\}$ then the state set of $H_3 = H_1 \Cap_\tau H_2$ is $\Sigma = \{\langle [5\text{–}6, 9\text{–}10), \text{Alibi}(P2, \text{“}\textit{Joan}\text{”})\rangle\}$, and the state set of $H_3 = H_1 \Cap_\sigma H_2$ is $\Sigma = \{\langle [1\text{–}6, 13\text{–}20), \text{Alibi}(P1, \text{“}\textit{Sarah}\text{”})\rangle\}$. They show the time periods when the two alibi's have common states based on either their common temporal elements or common snapshot values.

Definition 18 (Difference). The difference operation constructs a new history H_3 from two histories H_1 and H_2. Its result contains all elements of H_1 that are not present in H_2. There are again two versions of this operation: one, denoted by $\setminus\!\!\setminus_\tau$, tests for shared instants in the timestamp values; the other, denoted by $\setminus\!\!\setminus_\sigma$, tests for equality of snapshot values.

$$\begin{aligned}
&H_3 := \setminus\!\!\setminus_\tau (H_1, H_2) \equiv \\
&\quad H_3 := \texttt{Create()}\ (\forall s_1 \in S^{H_1}, \forall s_2 \in S^{H_2} | \underline{\texttt{common\_instants}}(s_1 \cdot \mathsf{t}, s_2 \cdot \mathsf{t})) \\
&\qquad [\texttt{InsertState}(H_3, \langle \underline{\texttt{minus}}(s_1 \cdot \mathsf{t}, s_2 \cdot \mathsf{t}), s_2 \cdot \mathsf{v}\rangle)]
\end{aligned}$$

$$\begin{aligned}
&H_3 := \setminus\!\!\setminus_\sigma (H_1, H_2) \equiv \\
&\quad H_3 := \texttt{Create()}\ (\forall s_1 \in S^{H_1}, \forall s_2 \in S^{H_2} | s_1 \cdot \mathsf{v} = s_2 \cdot \mathsf{v}) \\
&\qquad [\texttt{InsertState}(H_3, \langle \underline{\texttt{minus}}(s_1 \cdot \mathsf{t}, s_2 \cdot \mathsf{t}), s_2 \cdot \mathsf{v}\rangle)]
\end{aligned}$$

For example, if the state sets of two histories H_1 and H_2, representing the alibis of two suspects, are $\Sigma_1 = \{\langle [1\text{–}6), \text{Alibi}(P1, \text{“}\textit{Sarah}\text{”})\rangle, \langle [9\text{–}11), \text{Alibi}(P4, \text{“}\textit{Mary}\text{”})\rangle\}$ and $\Sigma_2 = \{\langle [5\text{–}10), \text{Alibi}(P2, \text{“}\textit{Joan}\text{”})\rangle, \langle [13\text{–}20), \text{Alibi}(P7, \text{“}\textit{Alison}\text{”})\rangle\}$ then the state set of $H_3 = H_1 \setminus\!\!\setminus_\tau H_2$ is $\Sigma_3 = \{\langle [1\text{–}5), \text{Alibi}(P1, \text{“}\textit{Sarah}\text{”})\rangle, \langle [10\text{–}11), \text{Alibi}(P4, \text{“}\textit{Mary}\text{”})\rangle\}$, and the state set of $H_3 = H_1 \setminus\!\!\setminus_\sigma H_2$ is $\Sigma = \{\langle [1\text{–}6), \text{Alibi}(P1, \text{“}\textit{Sarah}\text{”})\rangle, \langle [9\text{–}11), \text{Alibi}(P4, \text{“}\textit{Mary}\text{”})\rangle\}$. They show the periods when the alibi of the first suspect does not have common states with the alibi of the second suspect based on either shared instants or on equal snapshot values.

$$\mathtt{Dissect}^{I} : \mathbb{H} \to 2^{\mathtt{Instant} \times \mathbb{V}}$$

$$\mathtt{Dissect}^{T} : \mathbb{H} \to 2^{\mathtt{TimeInterval} \times \mathbb{V}}$$

Fig. 18. Example history mutator operations.

4.2.2.5. Mutator operations. Tripod provides a collection of operations that transform a given history into another collection that represents a particular *view* over the history. Such views can be utilized (for example) by the Tripod query calculus [10,15] to provide different ways of iterating over a particular history. For example, while some users will want to view a history in its previously described canonical form, other users may want to view a history as a collection of states that are timestamped with individual timestamp values to facilitate comparison of instants- and intervals-based histories. The collections returned by such operations is ordered according to the semantics of the operation and the nature of the resultant collection. For example, `Dissect` operations return a set of snapshot/timestamp pairs, with the timestamp consisting of *individual* (i.e., `Instant` or `TimeInterval`) timestamp elements. Their signatures are given in Fig. 18. Their definition now follows.

Given $H = \langle V, \mathtt{Instants}, \gamma, \Sigma \rangle$, $\mathtt{Dissect}^{I}(H)$ returns a `set` S of pairs of the form $\langle \tau', \sigma \rangle$, where $\sigma \in V$ and the type of τ' is `Instant`, obtainable as follows:

$S := \mathtt{Dissect}^{I}(H) \equiv$

$S := \{\} \cup (\forall \langle \tau, \sigma \rangle \in S^{H})[(\forall \tau' \in \underline{\mathtt{disassemble}}(\tau))[S := S \cup \{\langle \tau', \sigma \rangle\}]]$

The definition of the $\mathtt{Dissect}^{T}$ operation is analogous.

The views created by such mutator functions can be used by the Tripod spatio-historical calculus to provide different forms of iteration over a history. For example, domain generators over the collection monoids of the form $s \leftarrow H$ can be constructed, where $H \in \mathbb{H}$ and $s \in \mathbb{S}$, such that bindings for states s are drawn from a view H on a history. It is envisaged that developers may need to view a history as a simple (possibly gapped) time-series in which each state is characterized by a non-collection-based timestamp, i.e., a calculus expression ranges over states of the form $\langle s : \mathbb{V}, t : T \rangle$, where T is either `Instant` or `TimeInterval`. Such expressions will therefore utilize other generator functions, such as `Dissect`, to provide expressions such as $s \leftarrow \mathtt{Dissect}^{I}(H)$, where H is a `Instants`-based history. For example, if the state sets of a history H, representing the population history of a blocks is $\Sigma = \{\langle [1\text{–}5), 12000 \rangle, \langle [9\text{–}11), 14000 \rangle\}$, then $\mathtt{Dissect}^{I}(H)$ returns a set $S = \{\langle 1, 12000 \rangle, \langle 2, 12000 \rangle, \langle 3, 12000 \rangle, \langle 4, 12000 \rangle, \langle 9, 14000 \rangle, \langle 10, 14000 \rangle\}$.

The following section presents the Tripod OM and its associated ODL, focusing on the Tripod spatial and timestamp types and histories using our motivating example. It assumes knowledge of the ODMG Object Model, languages (i.e., ODL and OQL) at the level of [7].

4.3. Historical operations and run-time efficiency

The operations formally defined in the previous sections have been developed to exhibit satisfactory run time efficiency. Table 1 presents the run-time efficiency of a representative sample of historical operations, where n is the number of states in the history. In the developed system (see Section 7) these operations utilize efficient implementations of the Tripod timestamp types (based on the implementation of the ROSE types as described in [21] that utilize algorithms such as plain

Table 1
Time complexity of selected historical operations

Operation	Defined in	Time complexity
`InsertState`	Definition 2	$O(n)$
`DeleteState`	Definition 5	$O(n)$
`ContainsSnapshot`	Definition 8	$O(n/2)$
`FilterByTimestamp`	Definition 9	$O(\log n)$
`FilterBySnapshot`	Definition 10	$O(n/2)$
`Equals`	Definition 14	$O(\log n \times \log m)$
`Union`	Definition 16	$O(n + m)$
$\backslash\backslash_{\tau}$	Definition 18	$O(\log n \times \log m)$
$\backslash\backslash_{\sigma}$	Definition 18	$O(n \times m)$

sweep), and internal index structures to reduce the search space when iterating over one or more histories.

It should be noted that at first inspection the run time complexity of the `FilterByTimestamp` operation could be calculated as $O(\log n \times n)$, as the filtering step takes potentially $O(\log n)$ iterations (since we can index over timestamps as the key to the mapping from timestamps to snapshots defined by a history) and the internal call to `InsertState` takes potentially $O(n)$ iterations over the new history that retrieved states are inserted into. However, the internal call to `InsertState` has been optimized to zero time as (according to the constraints on histories presented in Section 4.2.1) we are assured that the snapshot of the state to be inserted will not be equal to that of any other states in the history that is being constructed.

`FilterBySnapshot` performs less well than the `FilterByTimestamp` since we can index over the key of the mapping defined by a history (recall that a history is defined as a mapping from timestamps to snapshot values), but do not index over the snapshots. An iterative method must therefore be employed in this case.

The `Equals` operation can utilize indexes defined on the timestamp keys of both its arguments (with state sizes n and m, respectively), hence providing $O(\log n \times \log m)$ performance.

The `Union` operation can also be optimized, since states in its second history argument, H_2, take precedence over those in its first history argument H_1, we can simply construct the history returned by the operation by first calling the copy constructor with H_1 as its argument, and subsequently call `InsertState` for each state in H_2. Thus giving $O(n + m)$ performance.

The Difference and Intersection operations provide two variants, one based on testing for equality of states on snapshots, the other based on timestamps. The performance of these variants differs considerably, as we can utilize indexes on timestamp keys in the timestamp variant, whereas an iterative method must be employed when evaluating snapshot values. These results can be generalized to all other history search/filter operations that involve either evaluation of either timestamp or snapshot values.

5. The Tripod spatio-historical object model

Tripod's historical extensions provide the ability to record the history of change that entities undergo over time, extending the ODMG object model with the ability to track the changes caused by assignment operations on any ODMG construct that is assignable and that is declared as historical. More specifically, histories make it possible to track, irrespective of their being spatial or aspatial, the changes caused by the creation and deletion of objects, assignments to object attributes, assignments to object relationships, and assignment to named collections. For example, in our crime application presented in Section 2, there is a need to track changes made to a block's boundary, the collection of buildings it encloses, the count of its population, and the block group it is a part of (amongst others).

For each ODMG Object Model concept that is value assignable, the Tripod OM provides a corresponding *historical* concept, as follows: historical atomic object types, historical attributes, historical relationships and historical collections.

The history construct does not appear within the Tripod OM as a distinguished collection type constructor, rather the keyword `historical` appears as a modifier within the ODL, indicating that a history should be maintained of the relevant concept. When such declarations are made, Tripod internally assigns instances of the history ADT for each historicised concept. This approach has been taken because if the history mechanism were to appear as a type, then declarations such as: `history<history<set<history<String>>>>` would be syntactically valid, in spite of their semantics being difficult to ascertain.

In addition to specifying that a database concept should have its history maintained, the application designer can also specify certain (defaulted) properties of the history, namely:

- *Granularity*—If a history has granularity γ, then there can be no more than one recorded snapshot for each granule of γ (the default is the chronon granularity [9]). Any attempt to assign a value whose granularity finer than γ will cause the value to be converted to granularity γ. It should be noted that (internally) the underlying temporal realm stores all timestamp values as chronon timestamps. Therefore, Tripod transparently maps coarser granularity timestamps to chronon timestamps.
- *Timestamp type*—Each change is associated with an `Instants` or a `TimeIntervals` timestamp. The default is `TimeIntervals`.

For example, in the Tripod ODL we can declare the classes representing our crime application as shown in Fig. 5 (note that only a subset of the classes are shown), where `DAY` and `YEAR` are reserved words denoting the corresponding granularities. The following sections define in detail the various constraints on each of the aforementioned historical concepts (Fig. 19).

5.1. Historical atomic object types

In the Tripod OM, any object type (i.e., any user-defined type) can be declared as historical. This declaration causes an extra attribute, whose value is a history and is referable in queries and application programs as `lifespan`, to be created and maintained by the type system for each instance of that type. The lifespan attribute records the *status* of the object for a particular `TimeIntervals` or `Instants` value in the modelled reality. The snapshot in each state of `lifespan` indicates whether the object was active or inactive for the specified timestamp. This capability allows objects to be de/reactivated within a particular database, as opposed to being simply created and deleted. For example, our crime application can create a new block object

```
struct Alibi
{  points location;
   string verifying_person;
};

struct Address
{  points location;
   string Street;
   long   number;
   string postcode;
};

historical(timeIntervals, DAY) class Block
( extent Blocks key id)
{  historical(timeIntervals, DAY) attribute string id;
   historical(timeIntervals, DAY) attribute regions geographical_extent;
   historical(timeIntervals, DAY) attribute long population;
   historical(timeIntervals, DAY) relationship Block_Group in_group
      inverse Block_Group::composed_of;
   historical(timeIntervals, DAY) relationship set<Building> encloses
      inverse Building::enclosed_by;
   relationship set<Crime_Incident> has_incidents inverse Crime_Incident::occurs_in;
   historical(timeIntervals, DAY) relationship set<Boundary_Feature> bounded_by
      inverse Boundary_Feature::bounds;

   // operations
   void split_block(in Regions county_boundary, out set<Block> new_blocks);
};

historical(timeIntervals, DAY) class Building
( extent Buildings key name)
{  attribute string name;
   historical(timeIntervals, DAY) attribute Address address;
   historical(timeIntervals, DAY) attribute regions geographical_extent;
   historical(timeIntervals, DAY) relationship Burglary undergoes inverse Burglary::is_of;
   historical(timeIntervals, DAY) relationship Block enclosed_by inverse Block::encloses;
};

class Crime_Incident
( extent Incidents)
{  attribute long crime_number;
   attribute Points location;
   attribute Instants date;
   historical(timeIntervals, DAY) relationship Suspect committed_by inverse Suspect::commits;
   relationship Block occurs_in inverse Block::has_incidents;
};

class Burglary extends Crime_Incident
( extent Burglaries)
{  historical(timeIntervals, DAY) relationship Building is_of inverse Building::undergoes;
   historical(timeIntervals, DAY)  relationship Administrative_Region in_admin
      inverse Administrative_Region::has_parcels;
};

historical(timeIntervals,DAY) class Suspect
( extent Suspects key name )
{  historical(timeIntervals,DAY)  attribute string name;
   historical(timeIntervals,DAY)  attribute Address address;
   historical(timeIntervals,DAY)  attribute set<Alibi> alibis;
   historical(timeIntervals, DAY) relationship set<Crime_Incident> commits
      inverse Crime_Incident::committed_by;
};
```

Fig. 19. Land use schema definition.

`bl:Block` to exist during the time period [10–20), and to be subsequently (logically) deleted during the period [12–15). The `lifespan` attribute for `Block` objects is internally defined as:

```
historical (TimeIntervals, DAY) attribute
  enum Status {active, inactive} lifespan;
```

which is populated in the example to give the history: `bl.lifespan` = {⟨[10–12, 15–20), `active`⟩, ⟨[12–15), `inactive`⟩}. Note that if an object is not created with an explicit timestamp then the default is the interval [`now`–`until_changed`).

If the designer chooses to declare an object type as non-historical, then this does not preclude the type having historical properties (although instances of the type will not have a `lifespan` attribute).

5.2. Historical properties

Any property (i.e., any attribute or relationship) of an atomic object type can be declared as historical. For example, the following statements (from Fig. 5) declare a historical attribute and a historical relationship for the `Building` atomic object type:

```
historical(timeIntervals, DAY) attribute regions geographical_extent;
historical(timeIntervals, DAY) relationship Burglary undergoes
 inverse Burglary::is_of;
```

In the ODMG object model, the integrity of a relationship is automatically maintained by the ODBMS. This is also the case with historical relationships, although such maintenance is inherently more complex. For example, when a new state is added to the `undergoes` relationship described above, the inverse relationship defined by the `Crime_Incident` type must be maintained so that the constraints defined in Section 5.5 are satisfied since both relationships have histories as values. Such maintenance is the responsibility of the Tripod database kernel. An example of a historical relationship is shown in Section 6.

5.3. Historical collection object types

An instance of a collection object type is a named object whose component elements are of the same type. This type can be an atomic object type, another collection, or a literal type. A historical collection object type utilizes the history mechanism to allow the history of change of its composition to be recorded. An example of such a collection could be the set of criminals that have previously been designated as "the most wanted". This collection would be declared as follows:

```
historical (timeIntervals, YEAR) list < Criminal > most_wanted;
```

where each state of this history is a structure $\langle \tau, \sigma \rangle$, where σ : `list < Criminal >` is a list of the most wanted criminals during the associated interval (in years).

5.4. Inheritance in a historical setting

Inheritance gives rise to several important considerations when defining specializations of types involving historical declarations. Since the intention of the notion of inheritance is to allow the *specialization* of a type by its subtype, thus providing more specific information in the subtype, it follows that in the Tripod OM, a subtype may only have a more specialized historical nature than its supertype. Therefore, a historical type cannot be specialized into a non-historical type, but a non-historical type can be specialized into a historical one.

The properties of a type may be added to by a subtype, or they may be refined according to some restrictions. For the reasons stated above, it is only possible for a non-historical property to be specialized to a historical property (on the same domain or a more specific one) in a subtype. The reverse is not possible since this will result in a loss of information in the subtype. The substitutability property of the ODMG object model requires that any attribute refined in a subtype is consistent when the subtype is cast in terms of its supertype. For example, if we declare a basetype `Person` to have a non-historical attribute `name`, and in its subtype `Suspect`, we specialize `name` to be historical (e.g., to maintain the history of the aliases that a suspect has used), then when a `Suspect` is viewed (cast) as a `Person` it must have a consistent (non-historical) value for its `name`. Since the value of a non-historical attribute is by definition valid at the present time, the Tripod OM, by default, accesses a historical type/property's snapshot at *now* in such circumstances. This function therefore utilizes the $\texttt{FilterByTimestamp\_}\underline{\texttt{equals}}(H, \texttt{now})$ operation on histories to compute the current value of a historical property. Note that this value could be empty if every value in the subtype history exists only in the past, in which case an exception is raised.

5.5. Historical containment constraints

The Tripod OM defines a set of historical containment constraints on historical objects and their properties. For example, the lifespan of a historical object instance must always contain the timestamps associated with each snapshot of each of its historical properties. If this were not the case then a database would contain historical information about the properties of an object that held when the object itself was not known to have existed. These constraints are defined as follows:

(a) For an instance o of an historical type, with a set of object valued historical attributes *attrs*, the lifespan of o must bracket the lifespan of each of its attributes.

$\forall a \in attrs,\ \texttt{Brackets}(o.\texttt{lifespan}, a.\texttt{lifespan})$ returns $\texttt{true}$

(b) For an instance o of an historical type, with a set of historical relationships *rels*, the lifespan of o must bracket the lifespan of each of its relationships.

$\forall r \in rels,\ \texttt{Brackets}(o.\texttt{lifespan}, r.\texttt{lifespan})$ returns $\texttt{true}$

(c) For a historical collection object O of type HCO, whose states are $\Sigma = \{s_1, \ldots, s_n\}$, the lifespan of HCO must bracket the timestamp associated with each state:

$\forall s \in \Sigma, \texttt{ContainsTimestamp\_}\underline{\texttt{surrounds}}(O.\texttt{lifespan}, s \cdot \texttt{t})$ returns $\texttt{true}$

Note that the above containment constraints apply when both concepts are historical. If this is not the case then the constraint does not apply.

6. An extended example application

This section presents examples of the Tripod OM in use in the context of the hypothetical crime database alluded to in Section 2 and (partially) specified in Section 5 of this paper. The implementation of this application requires that all objects are created and manipulated using Tripod's language bindings. This involves writing C++ programs to perform these tasks. In order to simplify syntax, all examples use pseudo-code that mimics how actual C++ programs manipulate application objects, making calls where appropriate to Tripod historical, spatial and timestamp operations. The functionality described in this example has been implemented as a prototype that is described in Section 7.

6.1. Interacting with the Tripod database

When interacting with the database through the language bindings, developers utilize automatically generated operations to set and get an object's properties. For example, for a `string` property called `name` Tripod automatically generates operations:

`void set_name(in string name)`, and

`string get_name()`.

For historical properties these are further *overloaded* to allow developers to either explicitly use a particular timestamp value or to default to the current time *now* when interacting with the history. For example, for a historical timeIntervals string property with granularity DAY, called `name`, Tripod generates:

`void set_name(in State < timeIntervals, string > s)`, and

`void set_name(in string s)` (which defaults to the current time), and

`history < timeIntervals, DAY, string > get_name()`.

Note that Tripod does not provide an overloading for the historical get operation because C++ does not permit overloading on return values, therefore all returned values are histories.

6.2. Creating objects

The first code example, in Fig. 20, illustrates the creation of objects representing blocks. Line 5 shows the creation of a new `Block` named **b314** whose valid time is the time period [1/1/1970–*until_changed*). Once created, the operation `split_block` (whose signature is given in Fig. 5, and whose operation body is presented in Fig. 22) is invoked on **b314** in line 9. The result of this operation is the (possible) partitioning of **b314** into *n* disjoint blocks, as the result of the creation of the county called *Baker*. The result of the operation is contained in the collection `resulting_blocks`. Line 14 illustrates (inside a try/catch block) how the violation of schema

```
 1 // create timeIntervals values for use throughout the example
 2 timeIntervals t1("1/1/1970 - until_changed");
 3
 4 // create block 314 to have valid time t1 and regions value r1 (shown in Figure 2)
 5 Block b314 := new(t1) Block("314", r1);
 6
 7 set<Block> resulting_blocks;
 8 // split the block with Baker's geographic extent
 9 b314.split_block(Baker.get_extent(), resulting_blocks);
10
11 // create a school to have a current valid time (defaulted) and
12 // regions value r2 (shown in Figure 4)
13 try {
14   Building aSchool := new Building("Baker Elementary", r2);
15
16   // insert aSchool into the appropriate block
17   // enclosed_by is a 1:N historical relationship
18   aSchool.enclosed_by := b314;
19 } catch(exception e) {
20   // do something sensible if exception caught
21 }
```

Fig. 20. Creating Tripod objects.

constraints can be detected, thus ensuring that the creation of a new building within a block does not cause the geographic extent of the building to overlap with any other existing building. This is enforced in the definition of the `Building` class's constructor function shown in Fig. 21. If the building is successfully created then line 18 adds a new state to the historical relationship between itself and **b314** by assigning the new value to the history. This assignment transparently causes the history `InsertState` operation to be invoked so that the inverse of this historical relationship is automatically maintained. Thus, `aSchool` will be added to **b314**'s collection of buildings that it encloses, as specified by the `encloses` relationship.

The operation to construct a new `Building` is shown in Fig. 21. This operation checks that there is no overlap between the geographic extent of any existing building by iterating over the collection of all buildings (line 4) and finds the building's current snapshot value using the history `CurrentValue` operation. If there *is* a current value and this value overlaps with the input `geo_extent` parameter then an exception is raised, otherwise the new data is added to the current object using its `set` methods. Note that the defaulted version of `set_geographic_extent` cannot be used because the valid time of <u>`this`</u> (the containing object) may not be [`now–until_changed`). Therefore, the new state `s` is created with the valid time of <u>`this`</u>, accessed using the <u>`valid_time()`</u> system-level operation as shown in line 21.

6.3. Maintaining historical containment constraints

The historical containment constraints discussed in Section 5.5 are automatically enforced by Tripod. Therefore, if an attempt is made to update an object's historical property with a state whose valid time is not contained within the `lifespan` of the object an exception is raised.

```
 1 void Building(in string name, in regions geo_extent) throws exceptio
 2 {
 3   // iterate over the collection of all Buildings
 4   for each b in Buildings
 5   {
 6     // find the current state of the buildings geographic_extent
 7     regions current_value;
 8     try {
 9       current_value := b.get_geographic_extent().CurrentValue();
10
11       // see if the new value to be inserted causes any problems
12       if(current_value.overlaps(geo_extent))
13       {
14         throw exception("Illegal building creation operation");
15       }
16     } catch(exception e) {
17       continue;
18     }
19   }
20   // all is ok, so insert the new data
21   State<timeIntervals, regions> s(this->valid_time(), geo_extent);
22   // geographic_extent is a historical attribute
23   this->set_geographic_extent(s);
24   // name is a non-historical attribute
25   this->set_name(name);
26 }
```

Fig. 21. Definition of the `Building` constructor function.

For example, if a building object were created as follows:

```
Building aSchool :=  new(timeIntervals("1/1/2000–1/3/2002"))
                              Building("BakerElementary", r2);
```

making the object's `lifespan` attribute: $\{\langle [1/1/2000–1/3/2002), \mathit{active}\rangle\}$, and line 23 in Fig. 21 were changed so that the default `set` method was invoked, as follows:

```
this → set_geographic_extent(geo_extent);
```

when `now = 1/1/2002` (thereby making the valid time of the object's geographic_extent equal to `[1/1/2002–until_changed)`), then this changed line 23 would cause an exception to be thrown. It is left to the application programmer to choose a policy for handling such exceptions, as indicated in line 20 of Fig. 20.

6.4. Invoking operations

As described in Section 2, occasionally a land block may need to be split/merged if, for example, its population grows to exceed the maximum permitted for that block, or if the update of a block

```
 1 void split_block(in regions county_boundary, out set<Block> new_blocks)
 2 {
 3   // initialise new_blocks to be the empty set
 4   new_blocks := {};
 5   timeIntervals ti_now("now - until_changed");
 6
 7   // get the current (now) value of the geographic extent, (note this should also
 8   // verify that the history has a value)
 9   regions current_value := this->get_geographical_extent().CurrentValue();
10
11   // only do anything if there is an overlap between county boundary
12   // and my geographic extent
13   if (county_boundary.overlaps(current_value))
14   {
15     // compute my new geographic extent
16     regions updated_gext = this->get_geographical_extent().intersection(county_boundary);
17     State<timeIntervals, regions> s(updated_gext, ti_now);
18     this->set_geographical_extent(s);
19
20     // compute my new identifier according to TIGER rules
21     this->set_id(this->get_id() + compute_affix());
22
23      // compute the area outside the county for the new blocks
24     regions new_blocks_gext := county_boundary.minus(this->get_geographical_extent());
25
26     // iterate over new blocks
27     for each a in new_blocks_gext.disassemble()
28     {
29       // create a new block with a new identifier
30       // Note that a is of type region, and therefore must first be converted to
31       // a regions value using the assemble operation.
32       Block new_block := new(ti_now) Block((get_id() + compute_affix()), assemble(a));
33
34       // insert new block into result collection
35       new_blocks.insert_element(new_block);
36     }
37   }
38 }
```

Fig. 22. Definition of the `split_block` operation.

group (such as a County) causes a split. The `split_block` operation shown in Fig. 22 illustrates several interesting features of both the Tripod spatial types and its historical constructs. For example, line 13 utilizes the `overlaps` spatial predicate to test if the current value of the current object's (referred to as `this`) geographic extent has an overlap with the input `county_boundary`. Note how we must use the `CurrentValue` historical operation on line 9 to filter the geographic extent's history to get the value that is valid at time `now`—it is possible that there may be no current value, so complete code should verify the results of the `CurrentValue()` operation. Line 17 explicitly creates a new state to insert into the object's geographic extent history with the valid time `[now–until_changed)`. Note we could have used the default version of the `set` operation as this would have defaulted to this timestamp anyway. Line 21 computes an object's `id` using the TIGER rules implemented in `compute_affix()`, and the id of the present block. If there is an overlap then line 24 computes the exterior of the overlap using `minus`, a Tripod spatial operation, resulting in a single `Regions` value that may contain several disjoint elements. Each of these elements becomes a new `Block` object (line 32) that is inserted into the return collection `new_blocks`.

6.5. Inserting states into a history

6.5.1. Historical attributes

The next example shown in Fig. 23 illustrates how a series of states can be added to the history of a suspect's alibis. This example illustrates how the method used by application programmer to assign and delete values to/from the historical attribute (`alibis`). The result of the first print statement is (with details of the points literals omitted):

$$\{\langle[1/1/2001\text{–}5/1/2001), \text{Alibi}(p2, \text{“}\mathit{Mother}\text{”})\rangle, \langle[5/1/2001\text{–}8/1/2001), \text{Alibi}(p1, \text{“}\mathit{Wife}\text{”})\rangle,$$
$$\langle[8/1/2001\text{–}\mathit{uc}), \text{Alibi}(p2, \text{“}\mathit{Mother}\text{”})\rangle\}$$

whereas the result of the second print statement is:

$$\{\langle[1/1/2001\text{–}2/1/2001), \text{Alibi}(p2, \text{“}\mathit{Mother}\text{”})\rangle, \langle[3/1/2001\text{–}5/1/2001), \text{Alibi}(p2, \text{“}\mathit{Mother}\text{”})\rangle,$$
$$\langle[5/1/2001\text{–}8/1/2001), \text{Alibi}(p1, \text{“}\mathit{Wife}\text{”})\rangle, \langle[8/1/2001\text{–}\mathit{uc}), \text{Alibi}(p2, \text{“}\mathit{Mother}\text{”})\rangle\}$$

Line 21 of Fig. 23 creates the object representing the robbery with reference number `1106` that took place on `2/1/2001` at location p6. This is a non-historical object that has an attribute that is an instants timestamp, representing the time that the robbery took place.

6.5.2. Historical relationships

In an ODMG-compliant object database, an update to a relationship causes the reciprocal update of the relationship's inverse. In Tripod, this behaviour is extended to historical relationships. This facility however is inherently more complex than in its non-historical counterpart, and requires the use and co-ordination of operations over related histories. For example, if we were to update `Block B1`'s `encloses` relationship (a 1:M relationship with inverse

```
 1 timeIntervals ti1("1/1/2001 - 6/1/2001");
 2 timeIntervals ti2("5/1/2001 - 12/1/2001");
 3
 4 Suspect biggs := new(timeIntervals("1/3/72 - until_changed")) Suspect("Bloggs");
 5
 6 State<timeIntervals, Alibi> st1(ti1, Alibi(p2, "Mother"));
 7 State<timeIntervals, Alibi> st2(ti2, Alibi(p1, "Wife"));
 8
 9 biggs.set_alibi(st1);
10 biggs.set_alibi(st2);
11 // causes valid time to be [now - until_changed), where now
12 // in this example is 8/1/2001
13 biggs.set_alibi(Alibi(p3, "Sister"));
14 biggs.get_alibi().print(); // print statement 1
15
16 // part of an alibi is withdrawn
17 biggs.get_alibi().DeleteTimestamp(timeIntervals("2/1/2001 - 3/1/2001"));
18 biggs.get_alibi().print(); // print statement 2
19
20 // non-historical object
21 Robbery aRobbery := new Robbery(1106, p6, instants(2/1/2001));
```

Fig. 23. Assigning states to the `Alibis` historical attribute.

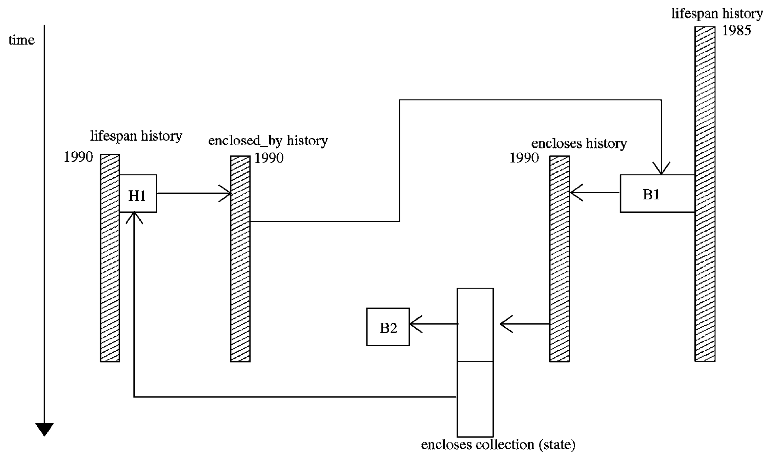

Fig. 24. Instantiated `encloses` relationship.

`Building::enclosed_by`), then this involves inserting a new state whose snapshot is a collection into `encloses`. Each element of this state's collection (i.e., a `Building`) must have its `enclosed_by` relationship set to `B1`. Although this is a complex operation for the database, the application programmer need only write the following code:

```
B1 → encloses.InsertState (new State
      × (d_TimeIntervals("1985–until_changed"), set(H1,H2)));
```

This results in `B1`'s `encloses` relationship containing a single state with a set-based snapshot value containing two elements, `H1` and `H2`, as illustrated in Fig. 24.

6.6. *Effects of updates to historical relationships*

If an object has all or a portion of its valid time deleted, then this can have ramifications for objects that have relationships with the deleted object. Deletion operations over historical attributes are simpler (but follow the same general method), as no propagation will take place. Note that in Tripod, relationships can only exist between object types and not between object and literal types. Furthermore, Tripod always maintains inverses to all object relationships.

An example of the 1:M relationship between 2 objects is shown in Fig. 24. If we were to delete `Block B1` during the time period 1990–1992, then this would have implications for (amongst other properties) the buildings that `B1` encloses at that time (as recorded through the object's `encloses` relationship). This deletion operation would be performed by a call to the following operation in Tripod's language bindings:

```
B1 → delete_object(d_TimeIntervals("1990–1992"));
```

Rather than deleting the entire object from the database, this operation results in an update of `B1`'s lifespan, indicating that the object has been *logically* deleted during this time period. Figs. 25–27 show how the deletion of this portion of `B1`'s lifespan is first propagated to the history that maintains the collection of buildings enclosed by the block, and is then subsequently propagated

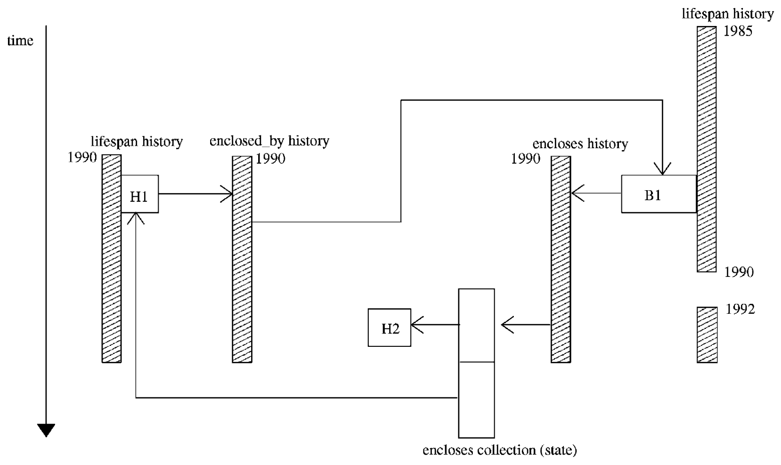

Fig. 25. Updating `B1`'s `object lifespan`.

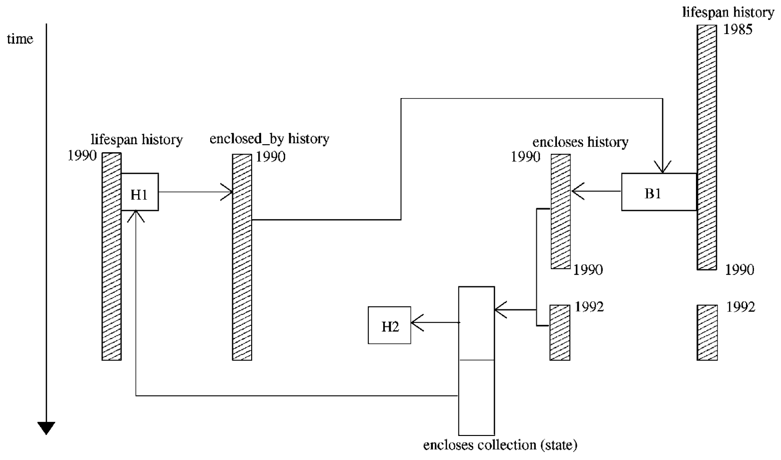

Fig. 26. Cascading to `B1`'s `encloses` history.

to the inverse of `B1`'s `encloses` relationship, `Building::enclosed_by`. These propagation operations are performed automatically by the database kernel to ensure that the object model's containment constraints (see Section 5.5) are satisfied. This functionality is provided for all historical relationship cardinalities (1:1, 1:M and M:M), once again providing application developers with kernel, rather than application-level functionality.

6.7. Querying a history through the language bindings

It is normal practice to issue queries over an object database using its declarative object query language (OQL), the results of which can be fed back to C++ programs for further analysis. Tripod does provide an OQL (suitably extended with timestamp, spatial and historical constructs), but its discussion is outside the scope of this paper (see [10,15–17] for further details). It is

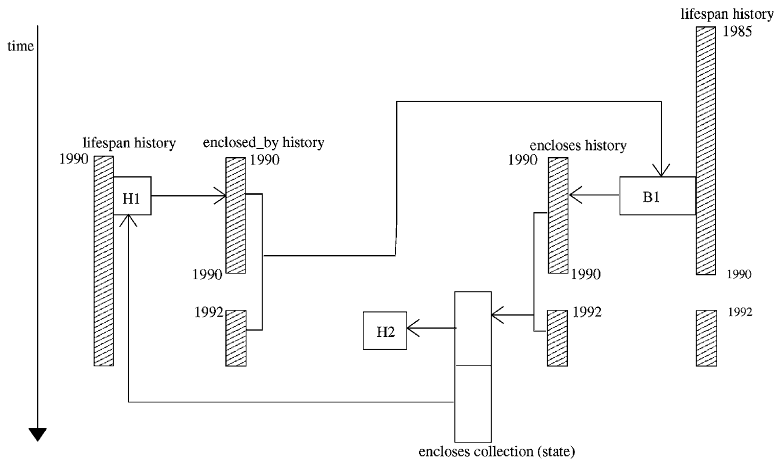

Fig. 27. Cascading to related Building's enclosed_by history.

also possible to retrieve data through the database's language bindings (i.e., C++). Fig. 28 shows two requests expressed in pseudo-code that compute the answers to the following questions:

(a) Who were the suspects that did not have an alibi for a given robbery?
(b) Who were the suspects that both were in the vicinity of a given robbery when it occurred and had an alibi for it?

These could be written in the Tripod OQL as follows:

Query (a) is answered in Fig. 28 by first obtaining the date of the robbery (line 5), and then iterating over all known suspects to access their alibis attribute. Line 12 calls the ContainsTimestamp historical operation instantiated with the common_points timestamp predicate to filter out all suspects that have an alibi for the time of the robbery. If a suspect does not have an alibi then they are added to the collection suspects_with_no_alibis. It should be noted that if this query were issued through Tripod's spatio-historical OQL, then the query would utilize any available indexes on the stored data, and hence improve the runtime efficiency of the algorithm.

Query (b) is answered by assuming that *in the vicinity* means within the block enclosing the location of the robbery. In line 20 we traverse Crime_Incident's non-historical relationship occurs_in to find the block that the robbery occurred in. We can then ascertain what the geographic extent (historical attribute) of that block was at the time of the robbery by calling the FilterByTimestamp historical operation instantiated with the common_points predicate as shown in line 23. The nature of the common_points predicate assures that the operation returns a history with either just a singleton value or that is empty. Because line 27 tests for the empty case, if that test fails we can use the LatestState operation to retrieve the single existing state. Once we have found the geographic extent of the block at the time of the robbery, we find all suspects that have an alibi for the time of the robbery by calling the ODMG-standard create_difference on set-valued collections (line 34). We then iterate over this collection and

```
 1 // Query (a)
 2 set<Suspect> suspects_with_no_alibis := {};
 3
 4 // get the date the robbery occurred
 5 instants ref_date := aRobbery.get_date();
 6
 7 // iterate over all suspects
 8 for each c in Suspects
 9 {
10    // alibis is a historical attribute
11    // check if c has no alibis for the crime's date
12   if(not c.get_alibis().ContainsTimestamp_common_instants_exists(ref_date))
13   {
14     suspects_with_no_alibis.insert_element(c);
15   }
16 }
17
18 // Query (b)
19 set<Suspect> suspects_with_alibis_in_locality := {};
20 Block robbery_block := aRobbery.occurs_in;
21
22 history<timeIntervals, DAY, regions> Hgext;
23 Hgext := robbery_block.get_geographic_extent().FilterByTimestamp_common_points(ref_date);
24
25 // FilterByTimestamp must return a history, even if it has only a
26 // single state. Check that the history is not empty.
27 if(not Hgext.isEmpty())
28 {
29   // The history will contain exactly 1 state, so use LatestState to get it.
30   try {
31     regions ref_gext := Hgext.LatestState().snapshot();
32
33     set<Suspect> suspects_with_alibis := Suspects.create_difference(suspects_with_no_alibis);
34
35     // iterate over all suspects with alibis
36     for each c in suspects_with_alibis
37     {
38       //iterate over each suspect's alibis history
39       for each s in c.get_alibis()
40       {
41         // if the alibis' location is in the same block as the crime
42         // add it to the result set.
43         if(s.snapshot().location.inside(ref_gext))
44         {
45           suspects_with_alibis_in_locality.insert_element(c);
46         }
47       }
48     }
49   }
50 }
```

Fig. 28. Answering queries through the language bindings.

over the states in the suspect's alibi history. Line 44 calls the `encloses` spatial predicate to check, in the alibi, whether the suspect was located in the block (note that each state of the alibis attribute is a structure with attributes `location` and `verifying_person`, as shown in Fig. 5). All qualifying suspects are added to the `suspects_with_alibis_in_locality` collection.

7. Implementation

This section describes in more detail the various components of the Tripod architecture shown in Fig. 1, and in particular (as shown in Fig. 29) how these components interact with each other in

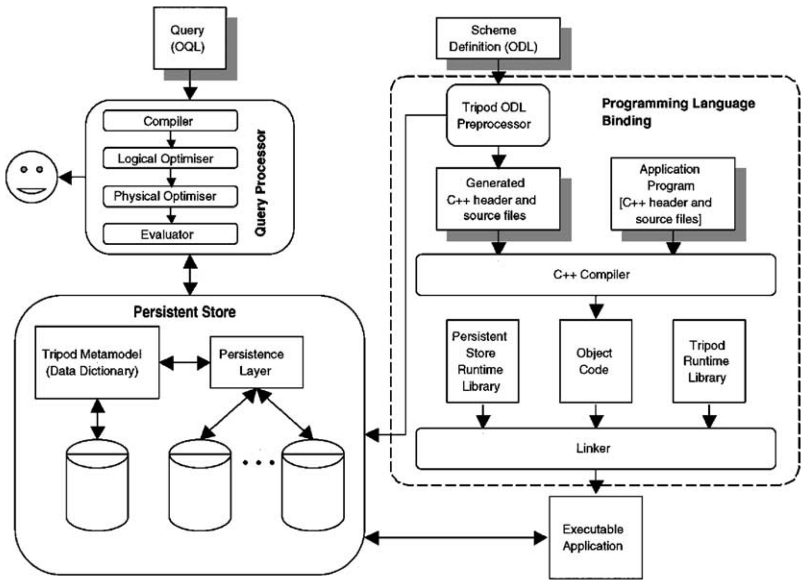

Fig. 29. Detailed Tripod architecture.

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.

The definition of a Tripod database consists of two parts: a schema (defined using a declarative object definition language (ODL)) to specify the structure of user-defined types and their behaviour, and an implementation of each of these behaviours specified using a programming language binding—in our case this is C++. Since the ODMG model does not define an object manipulation language (OML), developers must use the programming language binding to create, update and delete objects. This API corresponds to the operations presented in this paper to manipulate timestamp and spatial literal values and histories.

The Tripod ODL preprocessor lies at the core of the process of producing a database specification. It is responsible for analysing an ODL schema specification to produce: a set of C++ header files whose structure corresponds to that of the types expressed in the ODL schema definition; an instance of the Tripod metamodel (which is a superset of the ODMG metamodel) containing high-level information about the structure of the database schema that is used by (amongst others) the query processor; and methods to load and save persistent objects to and from the Tripod persistent store.

Once the application program and type information is compiled into object code, it is linked with libraries that implement the Tripod runtime system, and the persistent store. The library implements the core ODMG object model types as well as the Tripod spatial, timestamp and historical types. The persistent store runtime library contains the functionality needed to create and manage database connections, transactions and queries. The output of this process is an executable application that interacts with the underlying spatio-historical OODBMS.

When the state of a database needs to be queried, developers can either write native language application programs or issue declarative OQL queries. Tripod's OQL extends that defined by the ODMG with spatial, timestamp and spatio-historical constructs.

Interested researchers may download a prototype of the implementation from `http://img.cs.man.ac.uk/tripod/`.

8. Related work

There are relatively few proposals for temporal object models (see the survey in [40]), with the majority of previous research focussing on extensions to the relational model. Temporal extensions to the relational model have focussed on either tuple-timestamping [30,31] (each row in a table has an associated timestamp), or attribute timestamping [14] (each cell in a table can have an associated timestamp). The earlier tuple timestamping methods suffered from integrity problems when key columns were projected from relations, a problem that was not exhibited by the attribute timestamping methods. Although some implementation work was undertaken of tuple-timestamping models and their associated query language, we are not aware of any such effort with the attribute timestamping models.

With specific reference to temporal object models, the historical object model proposed by Bertino et al. [3] most closely relates to our work. They propose a temporal extension to the ODMG object model, called *T*_ODMG, that utilizes property timestamping to maintain how an object's properties change over time. The value for each *T*_ODMG object property is a function of time, utilizing interval-based timestamps that resemble the corresponding Tripod notion. It can be seen that Tripod provides additional modelling scope by providing instant-based timestamps, thus facilitating the modelling of histories with discrete states, capable of realising such properties as discrete temperature change over time. *T*_ODMG addresses concerns that we have not addressed, e.g., particular attention is paid to modelling objects that migrate to another type during their lifetimes. While the structural component of the *T*_ODMG model is well documented, the behavioural aspects of temporal domains are less so. We believe that the details of such behaviour (such as reported in this paper) are a necessary precursor to the definition of essential system components such as language bindings and a query language and its corresponding optimizer. It should be noted that in accordance with our aim of orthogonality, Tripod can function as a purely temporal database to provide functionality as described in this section.

There are several geographic information systems (GIS) available (e.g., [39]), and many database vendors are providing spatial capabilities in their systems (e.g., [1,34]), these systems however do not provide integral support for capturing how the spatial (or aspatial) properties of stored entities change over time. If a developer requires such facilities then they must write bespoke code to manage such time-varying data. It is this essential functionality that Tripod delivers to developers *within* the database kernel, thus removing the need for developers to write and optimize the required spatio-historical functionality. With specific reference to the spatial capabilities of existing GISs, we recognize that there will be functionality provided by these systems' operators that is not present in Tripod (the spatial features supported by commercial DBMSs is typically more limited than that of commercial GISs). In adopting the ROSE Algebra [23], we believe that we do however provide functionality that will capture the spatial requirements of the majority of spatial applications. Once again, Tripod can operate as a purely spatial database.

There are fewer proposals for spatio-temporal object models (see the survey in [43]). Of these the MADS model [36] reflects many of the concerns identified by the Tripod object model, including the orthogonal treatment of spatial and aspatial concerns. While the MADS model provides facilities to model spatio-temporal types and relationships, its orientation is towards the design of spatio-temporal applications, and therefore it currently does not directly consider the concrete operations necessary to manipulate and query actual data conforming to a specific MADS model. It is these types of applications that Tripod, through its spatial and historical constructs, realizes.

Both our data model and that of [3] utilize a *discrete* model of time. Other proposals exist for data models that capture objects whose properties (spatial and aspatial) are continuously changing. These models are typified by the *moving object* approach adopted in [22,37], which allow the state of each spatial and aspatial property to be expressed as a continuous function of time. Queries about the position of spatial data can then be inferred by the interpolation of spatial values between known bounds. This provides an expressive mechanism for the representation of moving points and polygons (although [37] only considers points). Querying moving object databases is achieved by extending an existing database algebra through a process called *lifting*. This allows non-temporal kernel algebra operations to be applied to temporal types. It should be noted, however, that such models do not provide comprehensive support for temporally changing aspatial data and object model constructs such as relationships, which are supported in a uniform way by the Tripod data model. In contrast, Tripod does not model continuous change, as it explicitly targets the large body of applications in which objects change in discrete steps. These include cadastral [42], cartographic [35], demographic [25], and crime-related [5] applications. The application in Section 2 utilizes both cartographic and crime-related features.

9. Conclusions

The aim of the Tripod project is to design and prototype a complete spatio-historical database system. This paper presents the core spatio-historical data model which forms the foundations of this work. In particular, this paper has:

(1) Presented a collection of primitive timestamp types whose foundations lie in the existing spatial ROSE algebra [23]. The intrinsic relationship between these timestamp and spatial types promotes consistent and complementary facilities for representing time and space.
(2) Described how these timestamp types can be used to underpin the notion of a history, as a generalized mechanism through which both spatial and aspatial change can be recorded over time.
(3) Shown how Tripod's spatial and timestamp types and the notion of a history can be used to orthogonally extend the ODMG object model to form a spatio-historical object model.
(4) Illustrated how these core modelling concepts are used within a spatio-historical database architecture to provide a formal description of the data structures and operations that are necessary to underpin both a query calculus and native language bindings.

(5) Shown the Tripod Object Model is use in an example drawn from the domain of crime analysis that illustrates the model in use.

Acknowledgements

Discussions with Michael Worboys, John Stell, Chris Johnson, Keith Mason, Nassima Djafri, Seung-Hyun Jeong and Bo Huang, our partners in the Tripod project, have contributed to shape the contributions of this paper. We are pleased to acknowledge our debt to them. This research is funded by the UK Engineering and Physical Sciences Research Council (EPSRC—grant number GR/L02692), whose support is gratefully acknowledged.

We would also like to thank the reviewers of the initial draft of this paper for their insights and valuable comments.

References

[1] PostgreSQL web pages, Web page, 2001. Available from <http://www.postgresql.org/index.html>.
[2] J.F. Allen, Maintaining knowledge about temporal intervals, CACM 26 (11) (1983) 832–843.
[3] E. Bertino, E. Ferrari, G. Guerrini, I. Merlo. Extending the ODMG object model with time. In: Proceedings ECOOP'98, 1998, pp. 41–66.
[4] C.R. Block (Ed.), Crime Analysis Through Computer Mapping, Police Executive Research Forum, 2001.
[5] P. Brantingham, P. Brantingham, Patterns in Crime, Macmillan Publishing Company, New York, 1984.
[6] P. Brantingham, P. Brantingham, Environmental Criminology, Waveland Press, 1990.
[7] R.G.G. Cattell (Ed.), The Object Database Standard: ODMG 3.0, Morgan Kaufmann, 2000.
[8] US Census, TIGER Home Page. Available from <http://www.census.gov/geo/www/tiger/>.
[9] C.S. Jensen et al., The consensus glossary of temporal database concepts, in: Temporal Databases: Research and Practice, Lecture Notes in Computer Science, vol. 1399, Springer, 1998, pp. 367–405.
[10] N. Djafri, A.A.A. Fernandes, N.W. Paton, T. Griffiths, Spatio-temporal evolution: querying patterns of change in databases, in: A. Voisard, S.-C. Chen (Eds.), Proceedings of the Tenth ACM International Symposium on Advances in Geographic Information Systems, McLean, Va, USA, November 8–9, ACM Press, 2002, pp. 34–41.
[11] FastObjects, Web page, 2003. Available from <http://www.fastobjects.com/>.
[12] L. Fegaras, D. Maier, Optimizing object queries using an effective calculus, ACM TODS 25 (4) (2000) 457–516.
[13] A.U. Frank, S. Grumbach, R.H. Güting, C.S. Jensen, M. Koubarakis, N.A. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H.-J. Schek, M. Scholl, T.K. Sellis, B. Theodoulidis, P. Widmayer, CHOROCHRONOS, A research network for spatiotemporal database systems, SIGMOD Rec. 28 (3) (1999) 12–21.
[14] S.K. Gadia, A homogeneous relational model and query languages for temporal databases, TODS 13 (4) (1988) 418–448.
[15] T. Griffiths, A.A.A. Fernandes, N. Djafri, N.W. Paton, A query calculus for spatio-temporal object databases, in: Proc. TIME, IEEE Press, 2001, pp. 101–110.
[16] T. Griffiths, A.A.A. Fernandes, N.W. Paton, S.-H. Jeong, N. Djafri, K.T. Mason, Bo Huang, M. Worboys, TRIPOD: A spatio-historical object database system, in: Mining Spatio-Temporal Information Systems, Kluwer Academic Publishers, 2002, pp. 127–146.
[17] T. Griffiths, A.A.A. Fernandes, N.W. Paton, K.T. Mason, B. Huang, M. Worboys, Tripod: A comprehensive model for spatial and aspatial historical objects, in: Proceedings of ER01, Springer-Verlag, 2001, pp. 84–102.
[18] T. Griffiths, A.A.A. Fernandes, N.W. Paton, T. Mason, B. Huang, M. Worboys, C. Johnson, J. Stell, Tripod: a comprehensive system for the management of spatial and aspatial historical objects, in: W.H. Aref (Ed.), Proc. 9th

ACM Int. Symposium on Advances in Geographic Information Systems (ACM-GIS), ACM Press, 2001, pp. 118–123.

[19] T. Griffiths, N.W. Paton, A.A.A. Fernandes, An ODMG-compliant spatio-temporal data model, preprint series, Dept of Computer Science, University of Manchester, 2000. Available from <http://pevepc13.cs.man.ac.uk/PrePrints/index.htm>.

[20] T. Griffiths, N.W. Paton, A.A.A. Fernandes, Realm-based temporal data types, preprint series, Department of Computer Science, University of Manchester, 2000. Available from <http://pevepc13.cs.man.ac.uk/PrePrints/index.htm>.

[21] R.H. Güting, T. de Ridder, M. Schneider, Implementation of the ROSE algebra: efficient algorithms for realm-based spatial data types, in: Proceedings of SSD'95, LNCS 951, Springer, 1995, pp. 216–239.

[22] R.H. Güting et al., A foundation for representing and querying moving objects, ACM Trans. Database Syst. 25 (1) (2000) 1–42.

[23] R.H. Güting, M. Schneider, Realm-based spatial data types: the ROSE algebra, VLDB J. 4 (2) (1995) 243–286.

[24] R.H. Güting, M. Schneider, Realms: a foundation for spatial data types in database systems, in: D. Abel, B.C. Ooi (Eds.), Proceedings of Advances in Spatial Databases—3rd International Symposium SSD'93, Singapore, June, LNCS, vol. 692, Springer-Verlag, 1993, pp. 14–35, ISBN 3-540-56869-7.

[25] L. Hattersley, R. Creeser, Longitudinal Study 1971-1991: History, Organization and Quality of Data. Number 7 in ONS Series LS. The Stationery Office, London, 1995.

[26] A. Hirschfield, K. Bowers (Eds.), Mapping and Analysing Crime Data, Taylor and Francis, 2001.

[27] P.B. Ladkin, The Logic of Time Representation. Ph.D. thesis, University of California at Berkeley, November 1987.

[28] G. Langran, Time in Geographical Information Systems, Taylor and Francis, 1992.

[29] M. Leipnik, D. Albert, D.H.J. Larmour (Eds.), GIS in Law Enforcement: Implementation Issues and Case Studies, Taylor and Francis, 2001.

[30] Nikos A. Lorentzos, The interval-extended relational model and its applications to valid-time databases, in: Theory, Design and Implementation, Benjamin/Cummings, 1993, pp. 67–91, ISBN 0-8053-2413-5.

[31] P. McBrien, Principles of implementing historical databases in RDBMS, in: Proceedings of the 11th British National Conference on Databases (BNCOD), July, Lecture Notes in Computer Science, vol. 696, Springer Verlag, 1993, pp. 220–237.

[32] V. Müller, N.W. Paton, A.A.A. Fernandes, A. Dinn, M.H. Williams, Virtual realms: an efficient implementation strategy for finite resolution spatial data types, in: 7th International Symposium on Spatial Data Handling—SDH'96, Taylor & Francis, 1996, pp. 697–710.

[33] National Institute of Justice, NIJ Mapping and Analysis for Public Safety Home Page. Available from <http://www.ojp.usdoj.gov/nij/maps/>.

[34] Oracle, Web page, 2003. Available from <http://www.oracle.com>.

[35] Ordnance Survey, The Digital National Framework, Web page, 2001. Available from <http://www.ordnancesurvey.co.uk/dnf/home.htm>.

[36] C. Parent, S. Spaccapietra, E. Zimányi, Spatio-temporal conceptual models: data structures + space + time, in: C.B. Medeiros (Ed.), ACM-GIS'99, Proceedings of the 7th International Symposium on Advances in Geographic Information Systems, ACM, 1999, pp. 26–33.

[37] D. Pfoser, C.S. Jensen, Y. Theodoridis, Novel approaches in query processing for moving object trajectories, in: VLDB 2000, pp. 395–406.

[38] Philadelphia Project, Philadelphia project web page. Available from <http://www.gis.com/specialty/government/philadelphia.html>.

[39] A.H. Razavi, ArcView GIS Developer's Guide, OnWord Press, 2001.

[40] R.T. Snodgrass, Temporal object-oriented databases: a critical comparison, in: Modern Database Systems: The Object Model, Interoperability and Beyond, Addison-Wesley/ACM Press, 1995, pp. 386–408.

[41] R.T. Snodgrass, Developing Time-Oriented Database Applications in SQL, Morgan Kaufmann Publishers, 2000.

[42] M. Talamo, F. Arcieri, G. Conia, E. Nardelli, Sicc: an exchange system for cadastral information, in: R.H. Güting et al. (Eds.), Advances in Spatial Databases, 6th International Symposium, SSD'99, Lecture Notes in Computer Science, vol. 1651, Springer, 1999, pp. 360–364.

[43] B. Theodoulidis, Review of spatiotemporal data models, Technical report, UMIST, 1998.
[44] D. Weisburd, T. McEwen (Eds.), Crime Mapping and Crime Prevention, Willan Publishing, UK, 1998.
[45] M.F. Worboys, A unified model for spatial and temporal information, Comput. J. 37 (1) (1994) 25–34.

Tony Griffiths is a Research Fellow with the Information Management Group in the Department of Computer Science at the University of Manchester. He was awarded an M.Sc. and Ph.D. in Computation by UMIST in 1993 and 1996, respectively. He has previously worked in the area of model-based systems, being a member of the team that designed and developed the Teallach system for automatically generating user interfaces to object databases. His principal interest lies in the design and implementation of spatio-temporal object databases, and their application to novel application areas. He also holds graduate and postgraduate degrees in music.

Alvaro A.A. Fernandes is a Lecturer in Computer Science at the University of Manchester. He was awarded an M.Sc. in Knowledge-Based Systems by the University of Edinburgh in 1989. From 1990 to 1995, at Heriot-Watt University in Edinburgh, he was a member of the team that designed and implemented the ROCK & ROLL deductive object-oriented database system. He was awarded a Ph.D. in Computer Science in 1995. In 1996 he was appointed for a lectureship at Goldsmiths College (University of London), where he worked on personalization and adaptivity mechanisms. In 1998 he joined the Information Management Group in Manchester. Besides Tripod, he has worked on MOVIE (an ODMG-compliant materialized view maintenance system), and on Polar* (a service-based ODMG-compliant distributed query processor over the Grid). His principal current interests are in spatio-temporal databases, distributed query processing in computational grids and logic-based approaches to knowledge discovery and their application to knowledge management, particularly the integration of deductive and inductive techniques in database settings.

Norman W. Paton is a Professor of Computer Science at the University of Manchester, where he co-leads the Information Management Group. He works principally on databases and distributed information management. Current activities include the development of distributed and service-based query processing technologies for the Grid in the OGSA-DAI and myGrid projects, and the design and implementation of spatio-temporal data models and query languages in the Tripod project. He also works on genome data management, in particular exploring the use of data integration techniques for making better use of functional genomic data. He is Co-Chair of the Database Access and Integration Services Working Group of the Global Grid Forum.

Dr. Robert Barr is Director of the Manchester Regional Research Laboratory and Senior Lecturer in Geographical Information Systems, in the School of Geography, University of Manchester. His research interests include geographical referencing, social information systems, spatial data warehousing, data mining and data policy issues. He is a past Chairman (2001) of the Association for Geographic Information (AGI) and chaired the AGI Conference committee for three years. He is a founding director of Manchester Geomatics, a geographic knowledge management company. Robert was a member of the Cabinet Office Social Exclusion Unit's "Better Information" Policy Action Team and chaired the geo-referencing sub group. He was a member of the United Kingdom Standard Geographic Base project board; has served in an advisory capacity for a number of government departments including the Ordnance Survey (the national mapping agency) and the Office for National Statistics. Robert speaks frequently at international conferences, has contributed regularly to the tes (Training European Statisticians Institute) courses for senior staff from statistical agencies throughout Europe and the Middle East. He is a long standing active member of URISA (the Urban and Regional Information Systems Association) and spent a year as a Harkness Fellow based at the National Center for Geographic Information and Analysis, University of California, Santa Barbara.