CS 332/780: Object-Oriented Databases Spring 2018
Course Notes #7
Keitaro Yukawa
Department of Computer Science
Queens College, CUNY

Modeling Temporal Information and History

In many application domains database systems are required to store temporal information associated with events and facts, sometimes along with their historical records. Recording history means that outdated or canceled data items will remain in the database, stamped with the time instants or periods of their validity, and will not be physically deleted or overwritten. Even in situations where such temporal information and history seem unnecessary for immediate purposes, recording them in the database might yield unforeseen payoffs later, for example for the purpose of data mining. As secondary storage capacities increase, more and more database systems are opting to record histories. Databases might also store temporal information about expected/predicted facts, events, and relations in the future, for example schedules of future events and probable trajectories of moving objects like airplanes and hurricanes.

Temporal Data Types and Relations

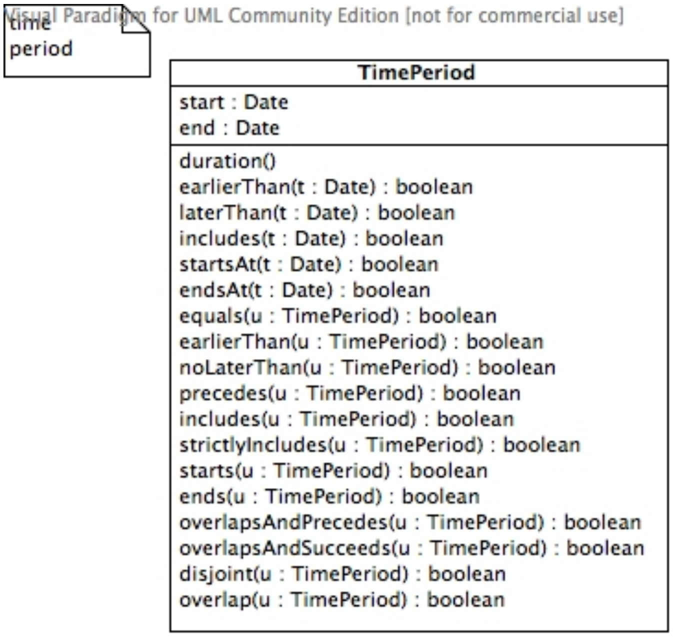
We assume that the *time instants* (*instants*, for short) are represented in some suitable way and are linearly ordered. A representation may be *discrete* or *continuous*. One possible discrete representation is by tuples of time units: …, microsecond, millisecond, second, minute, hour, day, month, year, … The smallest and largest units of tuples are determined by the nature of specific application domains. The smallest unit adopted in a discrete representation is called a *chronon*, representing one "tick" of a discrete clock. For example, the tuple format of (second, minute, hour, day, month, year), using the chronon of second, is suitable for many applications. An example of continuous representation is by real numbers indicating elapsed times, measured in some unit, from a certain fixed start time. The linear ordering will be denoted by < where $t_1 < t_2$ means t_1 is earlier than t_2. The relation $t_1 \leq t_2$ means $t_1 < t_2$ or $t_1 = t_2$, i.e., t_1 is no later than t_2. A *time period* (*period*, for short) is modeled by its start and end time instants, $[t_1, t_2]$, where $t_1 \leq t_2$. We assume that a temporal metric is defined to measure the length of $[t_1, t_2]$ in some unit, which will be called its *duration*. The following three special time instants are useful for temporal information processing. The instant *now* always represents the present time instant. The instants $-\infty$ and ∞ represent respectively indefinite past and indefinite future. We assume $-\infty < t < \infty$ for any instant $t \neq -\infty$, $t \neq \infty$.

The following is a list of basic binary relations over instants/periods useful for temporal operations and queries. Let t be any instant, T = $[t_1, t_2]$, U = $[u_1, u_2]$.

period/instant relation	definition
T is earlier than t	$t_2 < t$
T is later than t	$t < t_1$
T includes t	$t_1 \leq t \leq t_2$
T starts at t	$t_1 = t$
T ends at t	$t_2 = t$

period/period relation	definition
T equals U	$t_1 = u_1 \wedge t_2 = u_2$
T is earlier than U	$t_2 < u_1$
T is no later than U	$t_2 \leq u_1$
T precedes U	$t_2 = u_1$
T includes U	$t_1 \leq u_1 \wedge u_2 \leq t_2$
T strictly includes U	$t_1 < u_1 \wedge u_2 < t_2$
T starts U	$t_1 = u_1 \wedge t_2 < u_2$
T ends U	$u_1 < t_1 \wedge t_2 = u_2$
T overlaps and precedes U	$t_1 < u_1 < t_2 < u_2$
T overlaps and succeeds U	$u_1 < t_1 < u_2 < t_2$
T and U are disjoint	(T is earlier than U) $\vee$ (U is earlier than T)
T and U overlap	(T overlaps and precedes U) $\vee$ (T overlaps and succeeds U)

We assume there is a library class *Date* to represent time instants with attributes for recording times and dates, such as (second, minute, hour, day, month, year). The class *Date* is in fact provided in the *java.util* package and is supported by JDO/ObjectDB as a persistent type. Time periods are encapsulated as a class with the above binary relations coded by boolean functions:


Visual Paradigm for UML Community Edition [not for commercial use]
time period
TimePeriod
start : Date
end : Date
duration()
earlierThan(t : Date) : boolean
laterThan(t : Date) : boolean
includes(t : Date) : boolean
startsAt(t : Date) : boolean
endsAt(t : Date) : boolean
equals(u : TimePeriod) : boolean
earlierThan(u : TimePeriod) : boolean
noLaterThan(u : TimePeriod) : boolean
precedes(u : TimePeriod) : boolean
includes(u : TimePeriod) : boolean
strictlyIncludes(u : TimePeriod) : boolean
starts(u : TimePeriod) : boolean
ends(u : TimePeriod) : boolean
overlapsAndPrecedes(u : TimePeriod) : boolean
overlapsAndSucceeds(u : TimePeriod) : boolean
disjoint(u : TimePeriod) : boolean
overlap(u : TimePeriod) : boolean

Time Stamping, Valid and Transaction Times

The basic idea in recording temporal information is to *timestamp* data, events, and facts. Timestamping may involve:

- Single instant at which an event occurs.
- Set of instants at which an event occurs.
- Single period during which an event occurs or a data/fact/relation is valid.
- Set of periods during which an event occurs or a data/fact/relation is valid.
- Repetitive (cyclic) instant/period. Events occur, or facts/relations hold, repetitively or cyclically at specified instants or during specified periods. Specifications of repetition or cyclicity data are needed, like the time interval between successive time instants or between the start times of successive cycles. Sometimes repetition or cyclicity data is only implicitly or incompletely specified; this may be the case with schedules of events that take place in repetition, such as schedules of courses, trains, flights, movies, theatrical plays. For example a course schedule normally specifies incomplete periods (e.g., 5 – 6:15 PM, TU & TH), with the implicit understanding that the relevant course classes take place *each week* from the start of the semester through the end of it except for the days when the college is closed like holidays.

The times of events and facts in the real world may not coincide with the times that these events and facts are recorded in the database. For example, the time of a traffic accident is necessarily earlier than the times when its records are entered into the databases of a police department, motor vehicle department, and insurance company. In this regard, it is useful to distinguish two types of time:

- **valid time**: instant or period of the actual occurrence of an event or the validity of a fact in the real world;
- **transaction time (also called database time)**: instant or period when a record of an event/fact is entered into or stored in a database.

Models that maintain histories of valid and transaction times are called *bitemporal models*. Valid times are not necessarily earlier than transaction times. When schedules of future events are entered into databases, the start times of their transaction times are earlier than the valid times. Valid and transaction times may coincide, as when database systems receive input directly from events and timestamp them instantly. When students register for or drop courses online, the database system records the online transaction times of registrations which are also considered their valid times. Likewise, when we place purchase orders or do banking transactions online, their online transaction times are normally taken to be valid times as well. Events captured by sensors (like video cameras and audio recorders) and immediately recorded in databases would have nearly identical valid and transaction start time, with a slight delay of the transaction start time caused by signal transmission and processing in devices/systems.

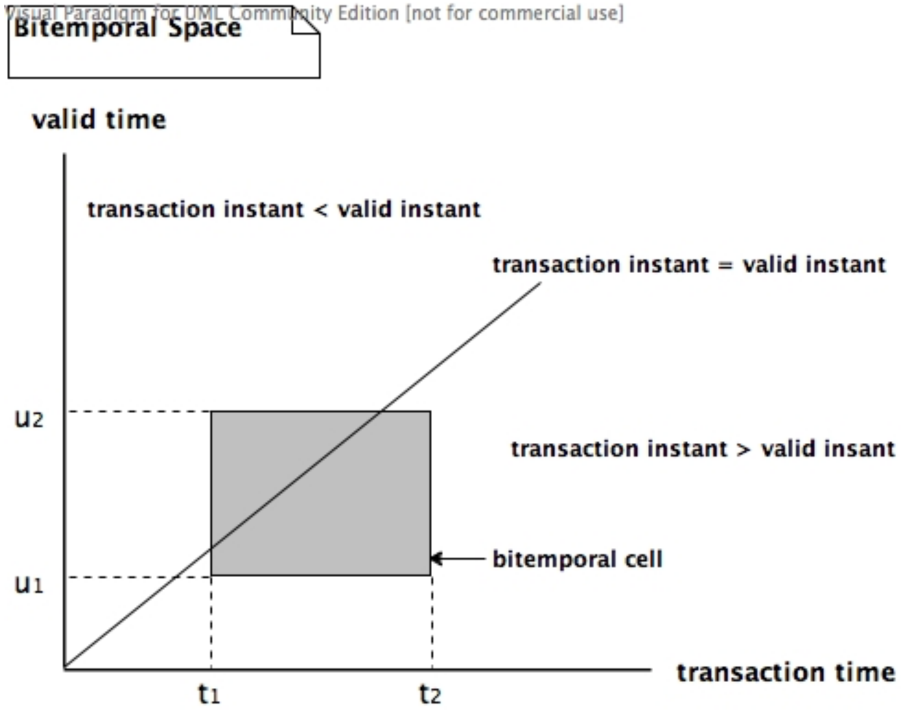
Central to maintenance of historical record is the concept of *logical deletion*: When an outdated/canceled data item is intended to be deleted or updated, the end of its transaction-time period is entered and the timestamped data item will physically remain in the database, becoming part of the historical record. Thus, whenever the database system finds the transaction end time of data stamped with a past instant, the data in question is interpreted as having been deleted or updated at that instant — this is the sense of "logical deletion". When an online account holder deletes or updates some of its data, such as credit card data and a phone number, the end instants of their transaction time periods are recorded and the data will remain in the database. The holder might even delete the entire account; again the entire account object will be timestamped with the end of its transaction time and will remain in the database. If a scheduled course section is canceled, the course section object would be marked as canceled and, if simultaneous deletion is intended, also logically deleted with a timestamp of its transaction end time.

Recall that we introduced two special instants: *now*, representing the present instant, and ∞, representing indefinite future with $t < \infty$ for any instant $t \neq \infty$. A valid time period [*t*, *now*] indicates that the associated data has been valid since instant *t* up to the present moment. A valid time period [*t*, ∞] indicates that the associated data is valid from *t* and expected to be valid until indefinite future; here *t* may be a past instant, *now*, or a future instant. A transaction time period [*t*, *now*] indicates that the associated data was entered into the database at *t* and no logical deletion has been done up to the present moment – such data is called *active*. Normally active data represent most up-to-date information about the real-world application domain. Any data item that was logically deleted or updated at *t′* should be stamped with a transaction time period [*t*, *t′*].

Bitemporal Space

In the following, transaction time instant and transaction time period will be abbreviated to transaction instant and transaction period; likewise, valid time instant and valid time period will be abbreviated to valid instant and valid period.

Aspects of relations between transaction and valid times can be captured geometrically in the *bitemporal space*, a 2-dimensional plane with transaction-time and valid-time coordinates.

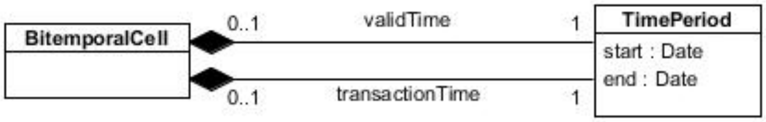

Visual Paradigm for UML Community Edition [not for commercial use]
Bitemporal Space
valid time
transaction instant < valid instant
transaction instant = valid instant
u2
transaction instant > valid insant
bitemporal cell
u1
transaction time
t1
t2

A data item timestamped with a transaction period [t_1, t_2] and a valid period [u_1, u_2] is associated with a rectangle called a *bitemporal cell*, defined by

$$[t_1, t_2] \times [u_1, u_2] = \{ <t, u> : t_1 \leq t \leq t_2, u_1 \leq u \leq u_2 \}.$$

The interpretation of this bitemporal cell is: at any transaction instant in [t_1, t_2], the database records that the associated data item is valid during [u_1, u_2]. If t_2 = *now*, the data item is *active*. If t_2 < *now*, the data item was logically deleted at t_2. In case $u_2 = \infty$, the bitemporal cell is open ended upwards. The diagonal line represents transaction instant = valid instant. The region below the diagonal represents transaction instant > valid instant, while the region above represents transaction instant < valid instant. For the example bitemporal cell displayed above, $u_1 < t_1 < u_2$ holds, meaning that when the data item was entered into the database at transaction instant t_1, valid start instant u_1 was in the past and valid end instant u_2 was in the future relative to t_1. At transaction instant t_2, both u_1 and u_2 were in the past relative to t_2. By drawing a vertical line at some transaction instant *t*, we can find data items along with their valid periods recorded in the database at *t*. By drawing a horizontal line at some valid instant *u*, we can find data items valid at *u* along with their transaction periods.

Bitemporal cells can be modeled as follows:



The following is a general procedure to update bitemporal history. When active data D with [t_1, *now*]×[u_1, u_2] is being updated at transaction instant t:

1. Logically delete D by changing its bitemporal cell to [t_1, t]×[u_1, u_2]. If discrete time is used, t may be replaced by its predecessor instant ("one tick before").
2. Associate D with the bitemporal cell [t, *now*]×(updated valid period of D).
3. If new data D' is replacing D, associate D' with [t, *now*]×(valid period of D').

Example Consider the following scenario of an employment history of Mr. Smith. The instants are represented discretely by tuples of (day, month, year) with the North American format month/day/year and with the chronon of day. In the description of facts, tense is used relatively to the respective transaction instants.

```
transaction time        fact and its valid time entered into database

   07/01/02:        Smith will work for Registrar's Office as transcripts manager
                    from 09/01/02 until 08/31/05.
   05/10/03:        Smith moved to payroll Office as assistant director on 04/01/03 and
                    will work there until 08/31/05.
   08/01/04:        Smith will move back to Registrar's Office as graduation auditor
                    on 09/01/04 and work there until 12/31/05.
   04/01/05:        Smith quit the job on 03/01/05.
```

If we use one-to-one correspondence between employment objects and bitemporal cells, the following six employment objects and the associated bitemporal cells have been created by 04/01/05:

07/01/02:

e_1(Smith, Registrar, transcr mgr) b_1 = [07/01/02, 05/09/03]×[09/01/02, 08/31/05] logically deleted

05/10/03:

e_2(Smith, Registrar, transcr mgr) b_2 = [05/10/03, now]×[09/01/02, 03/31/03] active

e_3(Smith, Payroll, assist director) b_3 = [05/10/03, 07/31/04]×[04/01/03, 08/31/05] logically deleted

08/01/04:

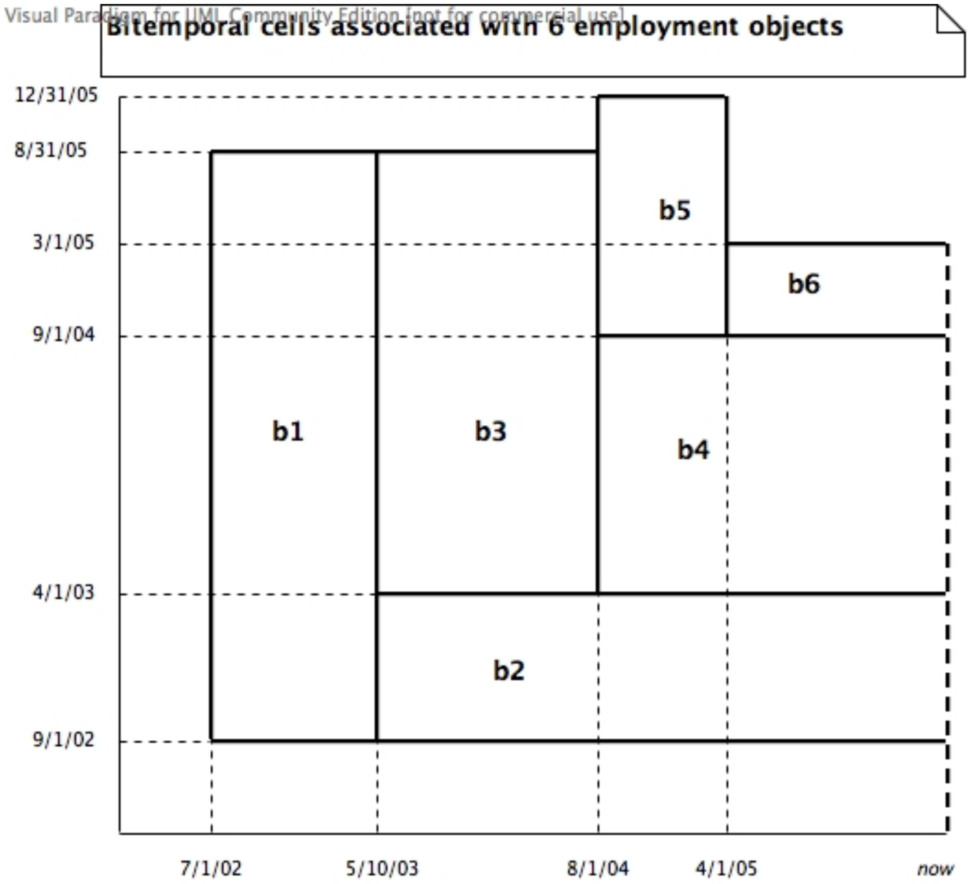
e_4(Smith, Payroll, assist director) b_4 = [08/01/04, now]×[04/01/03, 08/31/04] active

e_5(Smith, Registrar, grad auditor) b_5 = [08/01/04, 03/31/05]×[09/01/04, 12/31/05] logically deleted

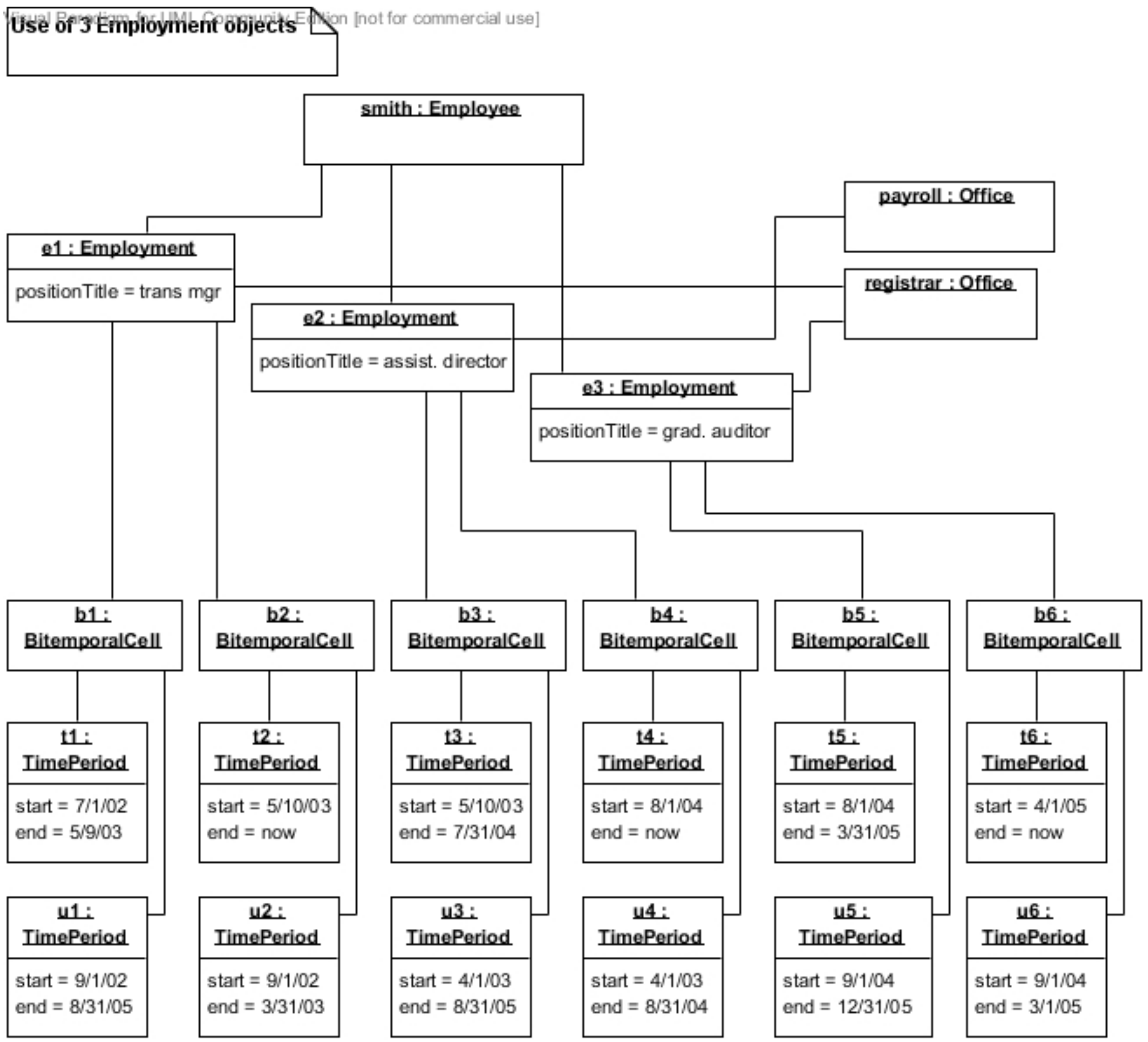
04/01/05:

e_6(Smith, Registrar, grad auditor) b_6 = [04/01/05, now]×[09/01/04, 03/01/05] active

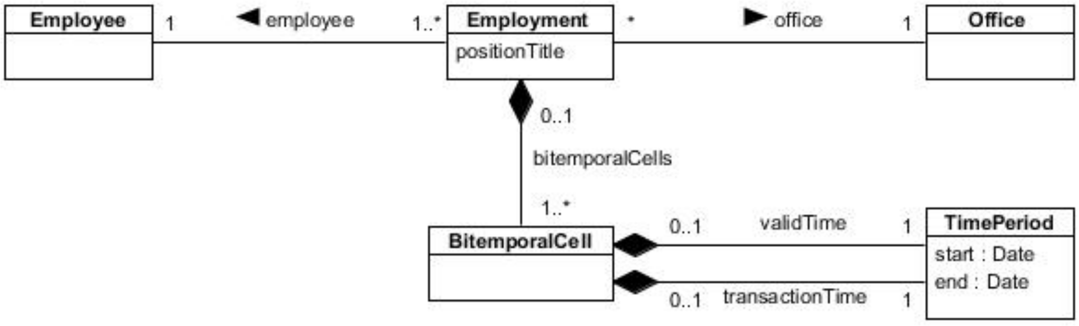
Note that the bitemporal cells with past transaction end instants indicate logical deletion, while those with the transaction end instant *now* indicate active data. The following bitemporal diagram displays six bitemporal cells b_i associated one-to-one with six employment objects e_i, 1 ≤ i ≤ 6.


Visual Paradigm for UML Community Edition [not for commercial use]
Bitemporal cells associated with 6 employment objects
12/31/05
8/31/05
3/1/05
9/1/04
4/1/03
9/1/02
b1
b2
b3
b4
b5
b6
7/1/02
5/10/03
8/1/04
4/1/05
now

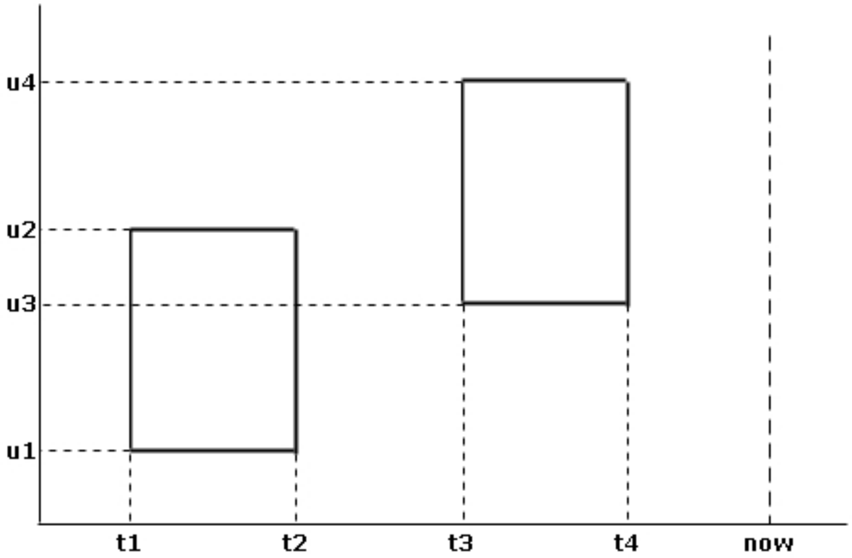
The model using six employment objects is workable but suffers from the same problem that plagued the tuple model of the Contract relation – an object with unique identity is split into two disconnected objects. The objects e_1 and e_2 represent Smith's first employment by Registrar's Office and should be represented by one employment object. Likewise, e_3 and e_4 represent the employment by Payroll Office and should be one object, and e_5 and e_6 represent the second employment by Registrar's Office and should be one object. The following object diagram illustrates a better model using three employment objects instead of six:



The model of the relevant classes is:



The transaction periods of an object/event may be non-contiguous and may not end at *now*. The database is considered to have no information/knowledge about an object/event during transaction periods without any associated bitemporal cells. For example, consider an event associated with the bitemporal cells below:



The database records during transaction period [t_1, t_2] that the event occurs during valid period [u_1, u_2]. At transaction instant t_2, this information is logically deleted. During transaction period [t_3, t_4], the database records that the event occurs during valid period [u_3, u_4]. This is logically deleted at transaction instant t_4. The database has no information/knowledge about this event during the transaction periods [t_2, t_3] and [t_4, *now*] except for historical record of outdated data.

If $u_2 < t_1$ and $u_4 < t_3$, valid periods [u_1, u_2] and [u_3, u_4] are in the past relative to t_1 and t_3, respectively. In this case, an interpretation of the above bitemporal diagram might be the following. The database records during transaction period [t_1, t_2] that the event is suspected to have occurred during valid period [u_1, u_2]. At transaction instant t_2, this information is logically deleted because of the detection of new conflicting evidence. At transaction instant t_3, new evidence emerges supporting the event's occurrence during valid period [u_3, u_4]. This is logically deleted at transaction instant t_4 because of the detection of newer conflicting evidence.

If $t_2 < u_1$ and $t_4 < u_3$, valid periods [u_1, u_2] and [u_3, u_4] are in the future relative to t_2 and t_4, respectively. In this case, an interpretation might be the following. The database records during transaction period [t_1, t_2] that the event is scheduled to occur during valid period [u_1, u_2]. At transaction instant t_2, this schedule is canceled and logically deleted pending a new schedule. At transaction instant t_3, a new schedule with valid period [u_3, u_4] is entered into the database. This is logically deleted at transaction instant t_4 because of the permanent cancelation of the event.

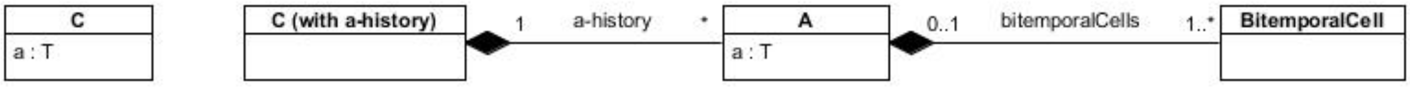
Turning now to general object-oriented bitemporal modeling methods, we identify three kinds of data that may be timestamped:

- object attributes
- objects
- relation instances

We will describe methods for timestamping these data and storing their bitemporal history.

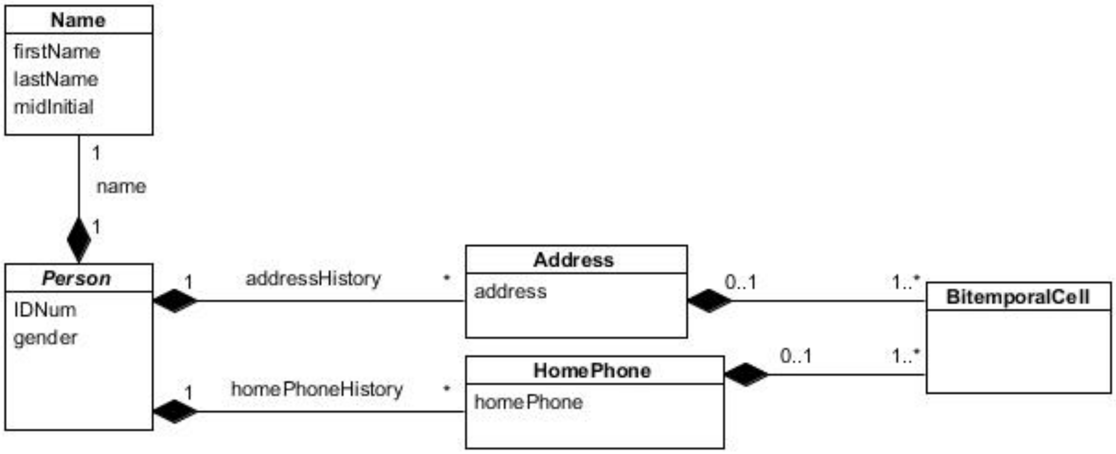
Timestamps for Object Attributes

Suppose a class *C* has an attribute *a* of type *T*, and we wish to record a bitemporal history of its values. This is accomplished by replacing the attribute *a* by the strong composition *a-history*: *C* → *A* where *A* is a subordinate attribute class with attribute *a*. The class *A* is then connected to the BitemporalCell class by strong composition:



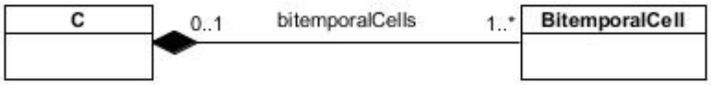
For each *C*-object *c*, a-history↓*A*(*c*) represents the history set of *a*-attribute values, each of which is stamped with bitemporal cells. The value of attribute *a* is never physically changed; an update of *a* to a new value is performed by creating a new A-object with the new value.

Consider the *Person* class in the college example. The following model records the bitemporal history of addresses and home phone numbers:

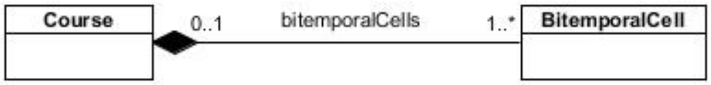


Timestamps for Objects

Class objects are timestamped by a strong composition relation to *BitemporalCell*:



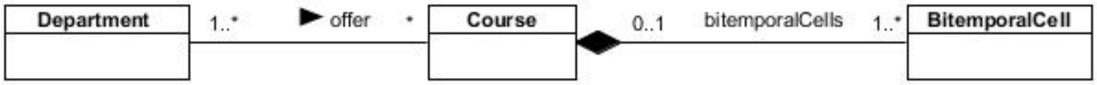
For example the valid and transaction times of *Course* objects can be recorded this way:



Here every Course object can only be logically deleted – it is never physically deleted. A valid period might indicate when a course becomes officially effective in the curriculum and when it is officially removed from the curriculum; the transaction time would record when a Course object is created, when updates of its valid period are done, and when it is logically deleted in the database.

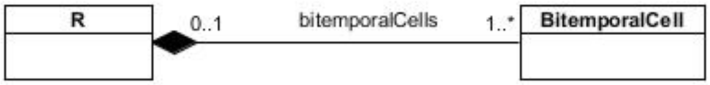
Timestamps for Relation Instances

In some cases the bitemporal histories of the instances of a binary relation R(A, B) are considered identical to those of associated B-objects. To be more specific: for each *a* ∊ A, *b* ∊ B such that the relation instance R(*a*, *b*) holds, the bitemporal history of R(*a*, *b*) is identical to that of *b*. Essentially, this occurs when each relation instance R(*a*, *b*) makes sense exactly when object *b* exists or valid. Consider the relation offer(Department, Course):



A department officially offers a course exactly when the course is officially effective in the curriculum, so the valid time of each *offer* relation instance is identical to that of the offered course. Also, it is reasonable to identify the transaction times of creation, updates, and logical deletions of every *offer* instance with those of the offered course. Another example is the relation manufactures(Manufacturer, Product) where the bitemporal history of every *manufactures* relation instance is set identical to that of the manufactured product. Therefore these kinds of binary relations do not need explicit timestamping, but it will be informative to state the identity of bitemporal history in suitable documentation areas.

In case the above identity does not hold, or the relation is 3-ary or higher, we need relation classes R to timestamp relation instances:



We saw an example of this in the above Employment relation. That is also an example where the valid times of argument objects in a relation can be defined by those of the associated relation objects: The valid periods of each Employee object can be defined by the union of the most up-to-date valid periods of all Employment objects in his/her employment history, which may not be contiguous, since the validity of employees is in effect precisely when their employment is in effect. For example, the valid period of Mr. Smith as employee is [09/01/02, 03/31/03] ∪ [04/01/03, 08/31/04] ∪ [09/01/04, 03/01/05] = [09/01/02, 03/01/05].

Temporal Sorting of History

The history of a set of objects is recorded by the set of bitemporal cells associated with them. It is convenient to store a set of objects and the set of bitemporal cells associated with each object in some temporal order to increase efficiency of temporal retrieval and queries. Since bitemporal cells are 2-dimensional, they can be sorted in different ways (i.e., there are different ways to define linear orderings on them), and the choice would depend on usefulness for a specific application domain.

One effective method of sorting a collection of bitemporal cells $[t_1, t_2] \times [u_1, u_2]$ is by lexicographic orderings based on lists of t_1, t_2, u_1, u_2 prefixed with signs + or −. The sign + indicates chronological order from past to future, while − indicates reverse chronological order from future to past. These lists will be called *sort signatures*. Each sort signature lists t_i and u_i in decreasing order of sort priority. For example, the sort signature $(+t_1, +u_1, -t_2, -u_2)$ indicates the following sorting method:

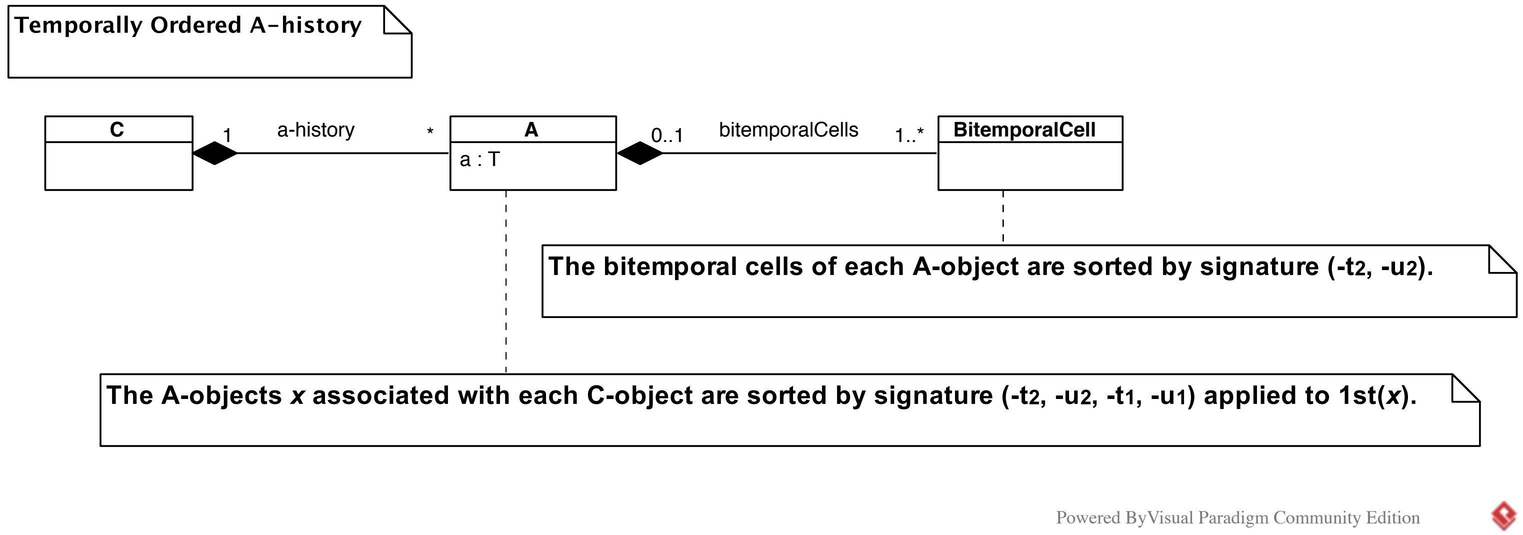
1. sort by t_1 in chronological order
2. if t_1 are equal, sort by u_1 in chronological order
3. if u_1 are equal, sort by t_2 in reverse chronological order
4. if t_2 are equal, sort by u_2 in reverse chronological order
5. if u_2 are equal, identical bitemporal cells are being compared: sort them in arbitrary order, or by an external criterion like values of an attribute of objects associated with bitemporal cells

Under some conditions, a set of bitemporal cells can be sorted by a signature of two or three elements. For example, assume that for each bitemporal cell $[t_1, t_2] \times [u_1, u_2]$, $t_1 < t_2$ and $u_1 < u_2$, i.e., no bitemporal cell degenerates into a line or point. Since the bitemporal cells associated with any *single* object do not overlap according to the bitemporal-history construction procedure (described before the Mr. Smith example), the signature of $(-t_2, -u_2)$ will completely sort the set of bitemporal cells of any single object (two bitemporal cells with equal values of t_2 and u_2 overlap). This holds true for every signature based on one of the four corners $<t_1, u_1>$, $<t_1, u_2>$, $<t_2, u_1>$, and $<t_2, u_2>$.

Under the above assumption, consider the Employment objects e_1, e_2, e_3 in the example employment history of Mr. Smith described earlier. By use of the signature $(-t_2, -u_2)$, the bitemporal cells of e_1 is sorted to b_2, b_1; those of e_2 is sorted to b_4, b_3; those of e_3 is sorted to b_6, b_5.

A set of objects may be sorted temporally by comparing the representative bitemporal cell selected for each object. The following is one possibility. For each object *x*, let 1st(*x*) be the first bitemporal cell of *x* in the temporal order chosen to sort the bitemporal cells of *x*. For a set S of objects, let $B_S = \{ 1st(x) \mid x \in S \}$. S may be then sorted by application of a sort signature to the bitemporal cells in B_S. For example, let S = $\{e_1, e_2, e_3\}$. Then 1st(e_1) = b_2, 1st(e_2) = b_4, 1st(e_3) = b_6, so $B_S = \{b_2, b_4, b_6\}$. By application of the sort signature $(-t_2, -u_2, -t_1, -u_1)$ to B_S, S is sorted to e_3, e_2, e_1. Since, in general, bitemporal cells in B_S may overlap, a full sort signature of four elements would be necessary.

In UML, notes attached to classes by dashed lines can be used to express requirements to be satisfied by their objects, including ordering requirements. The following model requires that for each *C*-object *c*, a-history↓*A*(*c*) be sorted in the above example temporal orderings:


Temporally Ordered A-history
C
1
a-history
*
A
a : T
0..1
bitemporalCells
1..*
BitemporalCell
The bitemporal cells of each A-object are sorted by signature (-t2, -u2).
The A-objects x associated with each C-object are sorted by signature (-t2, -u2, -t1, -u1) applied to 1st(x).
Powered ByVisual Paradigm Community Edition

The following is an example class schema of the *Person* class; rather than *Set*, the parametric *List* class is used to store the address and home phone histories in temporally sorted order:

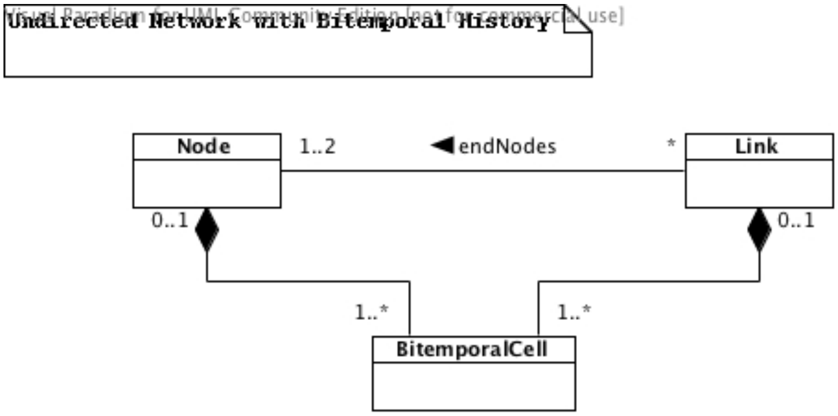
```
abstract class Person
{
        Name name;
        String IDNum;
        char gender;
        List<Address> addressHistory;  // sorted by signature (-t2, -u2, -t1, -u1) applied to
                                       // the set of 1st(a) where a is an Address object in this set
        List<HomePhone> homePhoneHistory;  // sorted by signature (-t2, -u2, -t1, -u1) applied to
                                           // the set of 1st(h) where h is a HomePhone object in this set

}

class Address
{
        String address;
        List<BitemporalCell> bitemporalCells;  // sorted by signature (-t2, -u2)
}

class HomePhone
{
        String homePhone;
        List<BitemporalCell> bitemporalCells;  // sorted by signature (-t2, -u2)
}
```

A parametric list type is available in JDO/ObjectDB as the parametric interface List<*C*> in the Collection<*C*> hierarchy; alternatively, the ordering requirement can be implemented by TreeSet<Address> and TreeSet<HomePhone> in the Collection<*C*> hierarchy, which permit more efficient retrieval with respect to ordered field values.

An alternative to storing objects in explicit temporal ordering by data types like *List* and *TreeSet* is to create sorted indexes (e.g., B-tree indexes) on transaction and/or valid times, which automatically and implicitly handle maintenance of objects in sorted order.

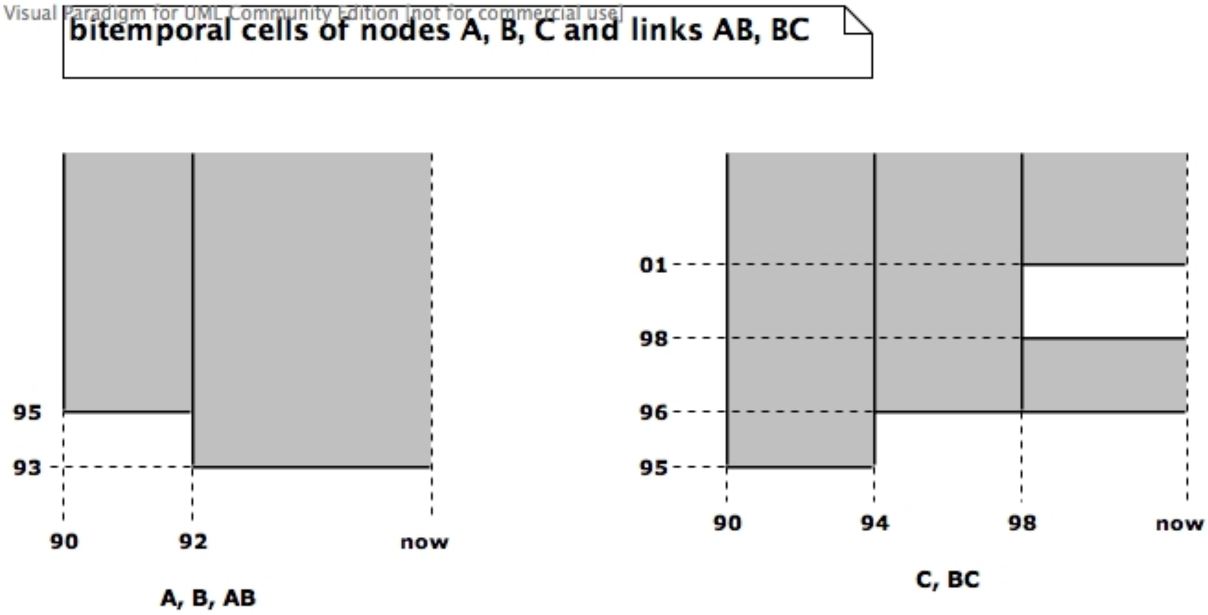
Network Example 1

The following model records bitemporal histories of undirected networks.


Visual Paradigm for UML Community Edition [not for commercial use]
Undirected Network with Bitemporal History
Node
1..2
endNodes
*
Link
0..1
0..1
1..*
1..*
BitemporalCell

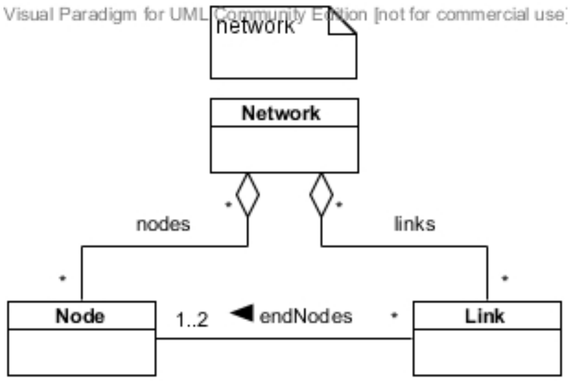
The following example scenario involves nodes A, B, C and links AB, BC. The chronon of year is used.

```
transaction time     fact and its valid time entered into database

   1990:           A, B, C, AB, BC are scheduled to open in 1995.
   1992:           A, B, AB are rescheduled to open in 1993.
   1994:           C, BC are rescheduled to open in 1996.
   1998:           C, BC are closed in 1998; they are scheduled to reopen in 2001.
```

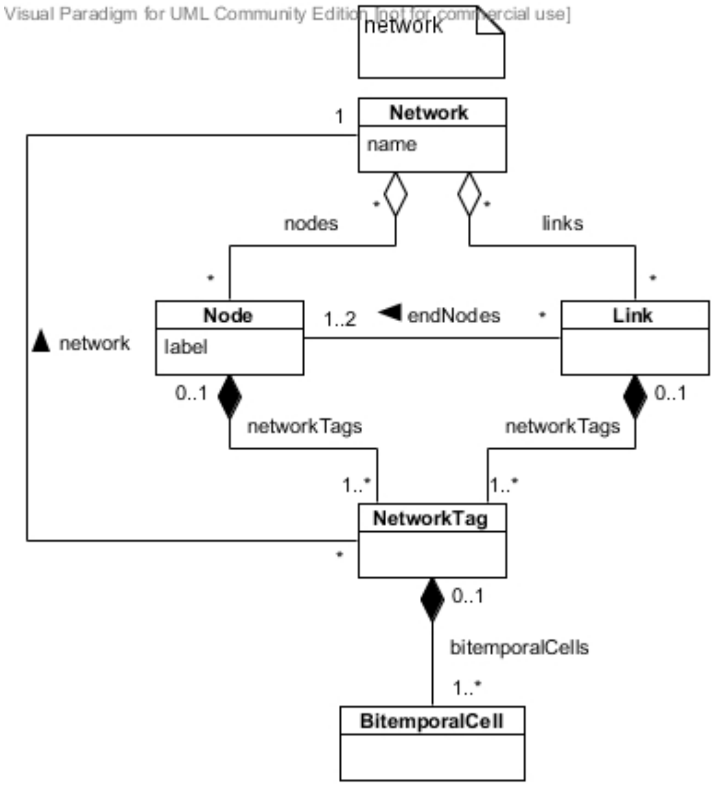

Visual Paradigm for UML Community Edition [not for commercial use]
bitemporal cells of nodes A, B, C and links AB, BC
95
93
90
92
now
A, B, AB
01
98
96
95
90
94
98
now
C, BC

Network Example 2

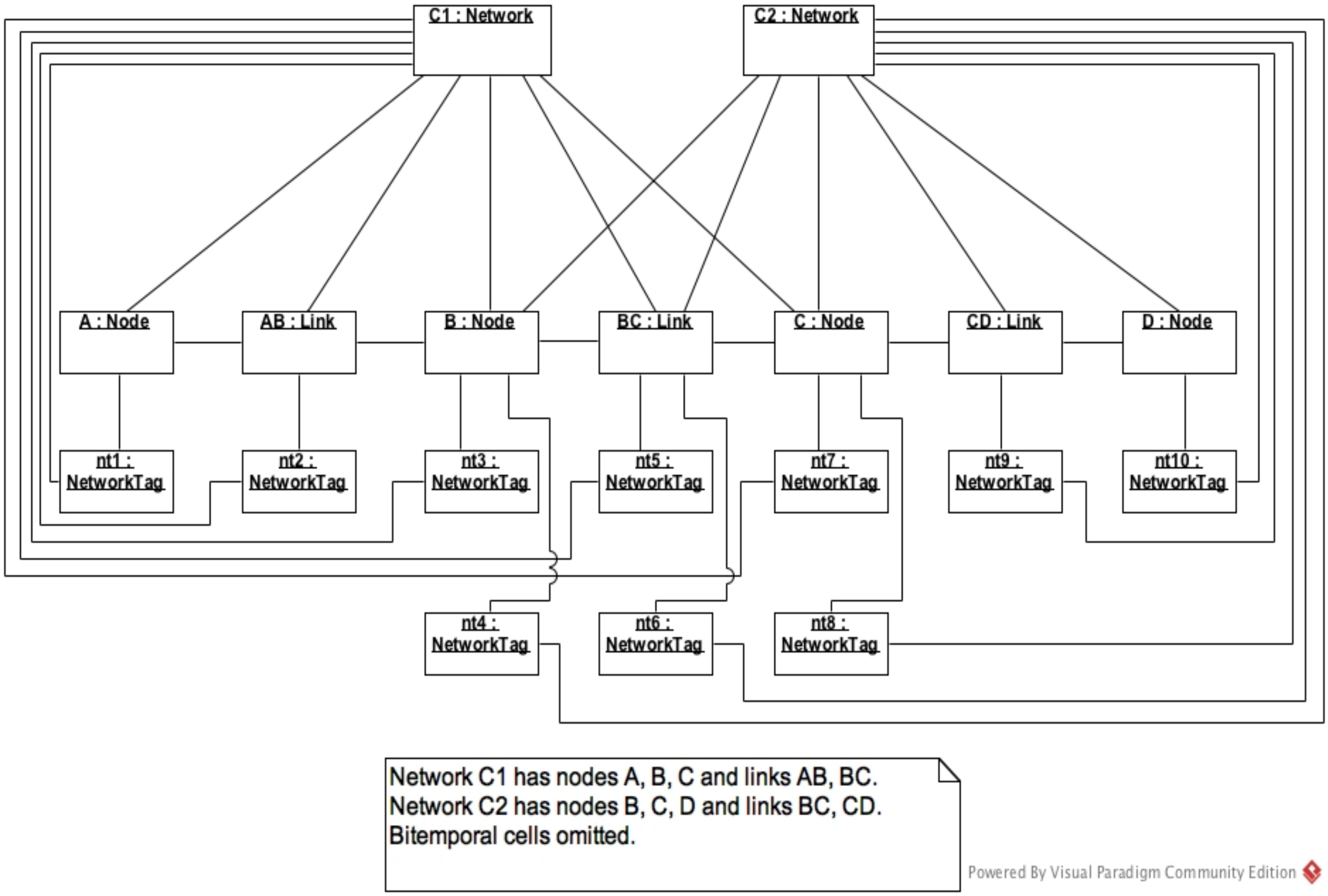
We now consider a model of networks where the same nodes and links may be parts of different networks and therefore have distinct bitemporal histories. The following is a basic model of undirected networks without temporal data:


Visual Paradigm for UML Community Edition [not for commercial use]
network
Network
*
*
nodes
links
*
*
Node
1..2
endNodes
*
Link

Here a node or link may be part of more than one network. For example, railroad stations/tracks and telephone stations/links might be shared by multiple networks operated by different companies or organizations. The following model is one way to handle these kinds of situations. Since a node or link may have distinct bitemporal histories for different networks, the model associates every node and link with network tags, each of which has the bitemporal histories of the node/link for the network identified by the *network* relation.



The following is an example where company C_1 operates a network of nodes A, B, C and links AB, BC, and company C_2 operates a network of nodes B, C, D and links BC, CD:



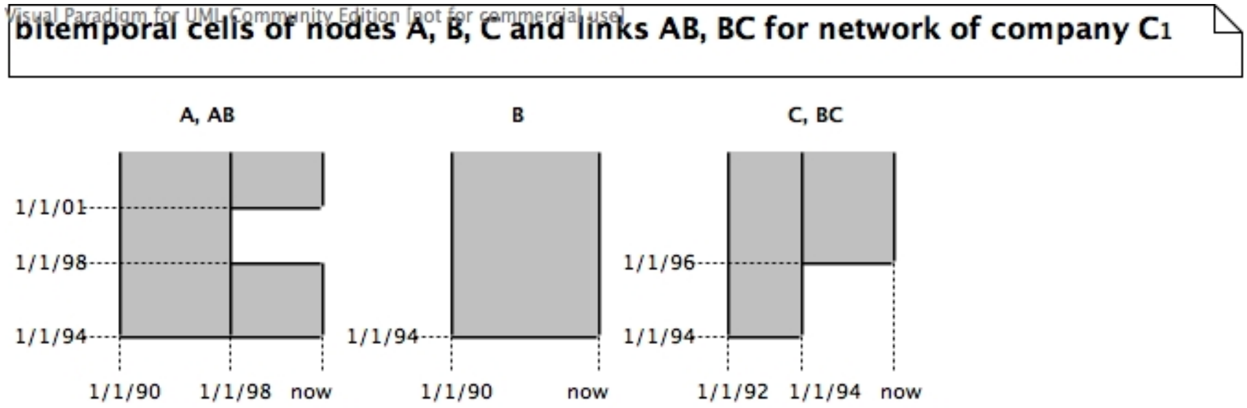
Network C1 has nodes A, B, C and links AB, BC.
Network C2 has nodes B, C, D and links BC, CD.
Bitemporal cells omitted.

Consider the following scenario of company C$_1$'s network:

> 01/01/90: Nodes A, B and link AB are scheduled to open on 01/01/94.
> 01/01/92: Node C and link BC are scheduled to open on 01/01/94.
> 01/01/94: The opening of node C and link BC is delayed until 01/01/96.
> 01/01/98: Node A and link AB are closed on 01/01/98, and are scheduled to reopen on 01/01/01.

The following bitemporal cells have been associated with the nodes and links by 01/01/98:

```
node A and link AB

01/01/90  [01/01/90, 12/31/97]×[01/01/94, ∞]
01/01/98  [01/01/98,      now]×[01/01/94, 12/31/97]
01/01/98  [01/01/98,      now]×[01/01/01, ∞]

node B

01/01/90  [01/01/90,      now]×[01/01/94, ∞]

node C and link BC

01/01/92  [01/01/92, 12/31/93]×[01/01/94, ∞]
01/01/94  [01/01/94,      now]×[01/01/96, ∞]
```


Visual Paradigm for UML Community Edition [not for commercial use]
bitemporal cells of nodes A, B, C and links AB, BC for network of company C1
A, AB
B
C, BC
1/1/01
1/1/98
1/1/94
1/1/90
1/1/98
now
1/1/94
1/1/90
now
1/1/96
1/1/94
1/1/92
1/1/94
now

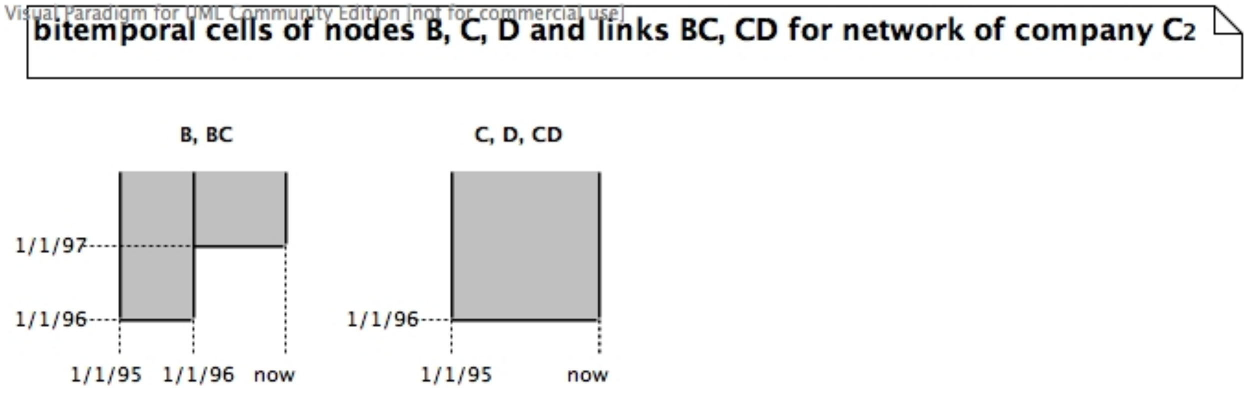
The above bitemporal cells of each node/link are parts of a single NetworkTag object that is related, via *network*, to a Network object operated by company C_1.

The next scenario is of company C_2's network:

> 01/01/95: Nodes B, C, D and links BC, CD are scheduled to open on 01/01/96.
> 01/01/96: The opening of node B and link BC is delayed until 01/01/97.

The following bitemporal cells have been associated with the nodes and links by 01/01/96:

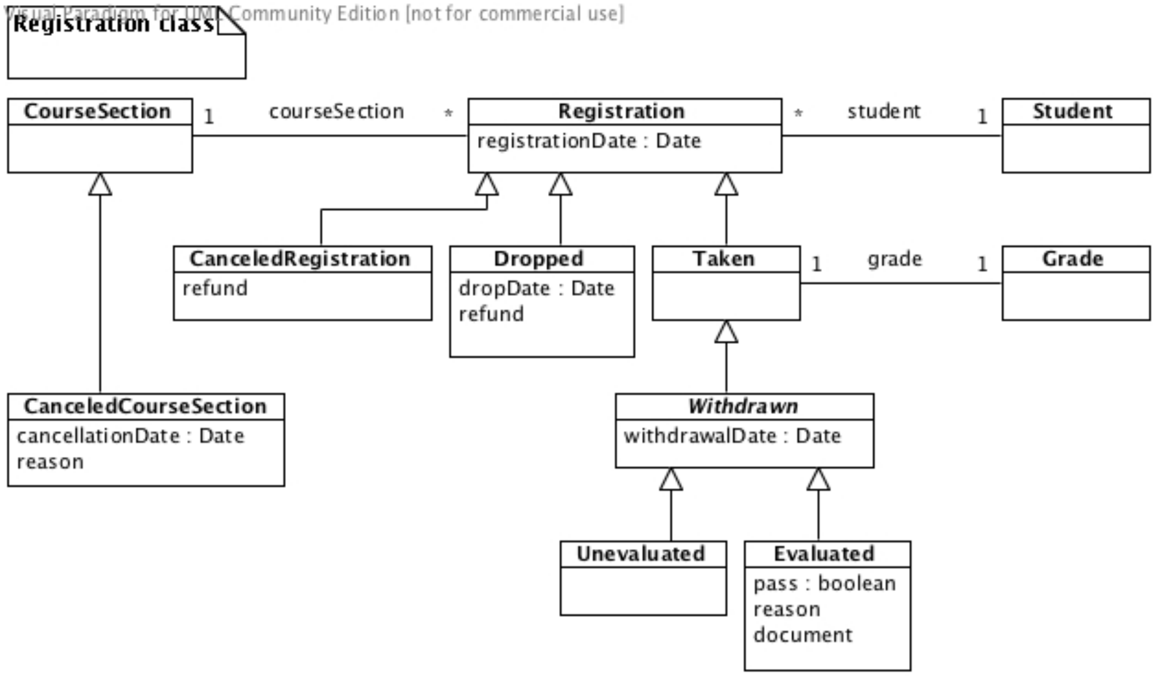
```
node B and link BC

01/01/95  [01/01/95, 12/31/95]×[01/01/96, ∞]
01/01/96  [01/01/96,      now]×[01/01/97, ∞]

nodes C, D and link CD

01/01/95  [01/01/95,      now]×[01/01/96, ∞]
```


Visual Paradigm for UML Community Edition [not for commercial use]
bitemporal cells of nodes B, C, D and links BC, CD for network of company C2
B, BC
C, D, CD
1/1/97
1/1/96
1/1/95
1/1/96
now
1/1/96
1/1/95
now

The above bitemporal cells of each node/link are parts of a single NetworkTag object that is related, via *network*, to a Network object operated by company C_2.

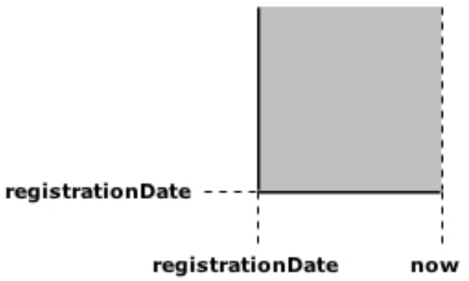
Registration Example

Let us refine the relation class *Take* in the college example to incorporate historical records of registrations. The following model is patterned after the system used by Queens College in 2008. Since the bitemporal cells of registration objects would be of very simple kinds, the model dispenses with bitemporal cells and records relevant valid/transaction instants in attributes. This example shows that a valid model can be built by adjusting the general methods to suit a particular situation rather than using them mechanically.

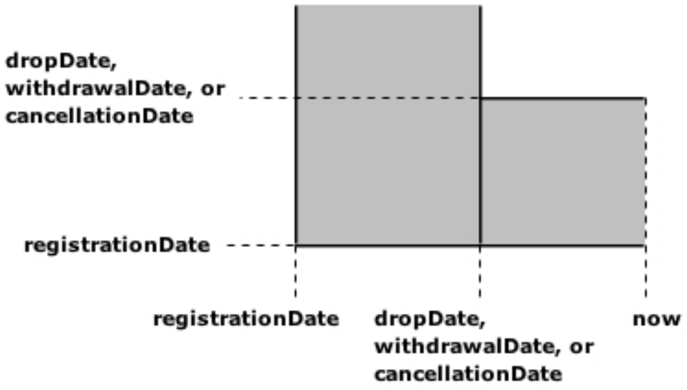


- Each new registration will be created in the *Registration* class.
- Up to the end of the first three weeks of a semester, each dropped registration will be moved to the *Dropped* class. No grade is assigned.
- During the 4th through 8th weeks, each withdrawn registration will be moved to the *Unevaluated* class. A grade of *W* is assigned.
- During the 9th week through the last day of classes, each approved withdrawn registration will be moved to the *Evaluated* class. A grade of *W* or *WF* is entered according as *pass* or *fail* is assigned by the instructor.
- At suitable times after the last day of classes and before final course grades entry, the objects remaining in *Registration* will be moved to *Taken*.
- When a *courseSection* is canceled, it is moved to *CanceledCourseSection*, and the *Registration* objects related to it, by *courseSection*, are moved to *CanceledRegistration*.

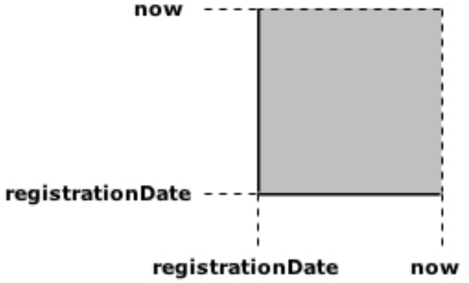
In this model the characteristics of the subclasses accrue partly from their temporal properties. *Registration.registrationDate* is the valid start instants of all registration objects. *Dropped.dropDate*, *Withdrawn.withdrawalDate*, *courseSection.cancellationDate*, and *now* signal the valid end instants of the objects in *Dropped*, *Withdrawn*, *CanceledRegistration*, and *Registration*, respectively. The registrations that have not been dropped, withdrawn, or canceled, stored in *Taken*, remain valid indefinitely, i.e., the valid end instant ∞ is implicitly assumed. The transaction times are not explicitly recorded. The transaction start instants of the registration objects are considered identical to their valid start instants as described. The transaction end instants are implicitly assumed to be *dropDate*, *withdrawalDate*, *cancellationDate*, or *now*. The diagram below depicts the bitemporal cells.



Objects in Taken



Objects in Dropped, Withdrawn, or CanceledRegistration



Direct Objects in Registration (Later become objects in Taken, Dropped, Withdrawn, or CanceledRegistration)