Modeling and Querying Current Movement : MOST data model and FTL query language (DOMINO)

The Problem

- Manage in a database the positions of a collection of objects moving around *right now*.
- Ask questions about current and near future positions and relationships between moving and static objects.

Lecture Outline

- Location Management
- MOST - A Data Model for Current and Future Movement
- FTL - A Query Language Based on Future Temporal Logic
- Evaluating FTL queries

Location Management

Various applications need to keep track of locations of moving objects.

- Taxi-cabs in a city: “Retrieve the 3 free cabs closest to Königsallee 48”
- Trucking company: “Which trucks are within 10 kms of truck T68 (which needs assistance)”
- Military: “Retrieve the friendly helicopters that will arrive in the valley in the next 30 minutes and then stay there for at least 15 minutes”.
- Location-based services: “Monitor the nearest motel to my (continuosly moving) truck”

Normally data in a database is constant unless explicitly updated.

Sending frequent position updates too costly. Not feasible for a large number of objects.

Basic idea: manage motion vectors. Positions change continuously without explicit updates.

Motion vectors not visible to the user. Instead, concept of **dynamic attribute** (MOST data model)

With dynamic attributes, we can ask queries about the future → language FTL (future temporal logic).

MOST - A Data Model for Current and Future Movement

Basic Assumptions

- Database: set of objects
- Each object belongs to an object class
- Each object class is given by a set of typed attributes
- Spatial data types: object classes have a spatial attribute (e.g. point, polygon) and are then called spatial object classes.
    - operations applied to objects rather than attribute values
    - *distance*(p_1, p_2), with p_1, p_2 point objects, not values of type *point*
    - *inside*(p, pol)
- Three kinds of spatial objects: spatial object class is a point class, a line class, or a polygon class.
- Special object called *Time* in database. Time discrete (*int*); value increases by each *clock tick*.

MOST: Dynamic Attributes

New idea of this approach: **dynamic attributes**.

Each attribute of an object class is either **static** or **dynamic**.
A dynamic attribute changes its value with time automatically.

```
car (license_plate: string, pos: (x: dynamic real, y: dynamic real))
```

Dynamic attribute A of type T (denoted A: T) represented by three subattributes

- $A.value$: of type T
- $A.updatetime$: a time value.
- $A.function$: f: $\underline{int} \rightarrow T$ such that $f(0) = 0$

Type T must have a value 0 and an addition operation.

Semantics:

$$value(A, t) = A.value + A.function(t - A.updatetime) \qquad \text{for } t \geq A.updatetime$$

Query refers to A: dynamic value is meant.

Query can refer to subattribute : E.g. find objects with pos.x.function = 5 (meaning $f(t) = 5\,t$).

MOST: Representing Object Positions

More realistic: Objects move along road networks.

Second way of modeling: Dynamic attribute *loc* with five subattributes:

- *loc.route*
- *loc.startlocation*
- *loc.starttime*
- *loc.direction*
- *loc.speed*

Semantics:

- A position (x, y) in the plane and on the network.
- At time *loc.starttime* it is *loc.startlocation*.
- At time *loc.starttime* + t it is the position on *loc.route* at distance *loc.speed* * t from *loc.startlocation* in direction *loc.direction*

MOST: Database Histories

Needed to define semantics of queries. Classical queries refer only to current state of the database.

A **database state** is a mapping that associates

- with each object class a set of objects of the appropriate type and values of their attributes
- with the *Time* object a time value

Notations

- *o.A* - attribute *A* of object *o*.
- *o.A.B* - subattribute *B* of ...
- *s*(*o.A*) - the value of *o.A* in state *s*
- *s*(*Time*) - the time of state *s*
- dynamic attribute A: its value in state *s* is *value*(*A*, *s*(*Time*))

A **database history** is an infinite sequence of database states, one for each clock tick. Starts at some time *u*.

$s_u, s_{u+1}, s_{u+2}, \ldots$

Value of attribute A in two states s_i, s_{i+1} can be different either due to an explicit update of A, or because A is a dynamic attribute whose value has changed implicitly.

An explicit update at time $t > u$ affects all states from t on. Old history H_u:

$$s_u, s_{u+1}, s_{u+2}, \ldots, s_{t-1}, s_t, s_{t+1}, s_{t+2}, \ldots$$

New history:

$$s_u, s_{u+1}, s_{u+2}, \ldots, s_{t-1}, s'_t, s'_{t+1}, s'_{t+2}, \ldots$$

Hence

- each clock tick: a new database state
- each explicit update: a new database history

Types of Queries

Queries are predicates over database histories. Distinction:

- instantaneous query
- continuous query

Instantaneous Query

Queries have implicit notion of current time. For example "within the next 10 time units".

Notation: $Q(H, t)$ - query Q evaluated on database history H assuming a current time t.

Query Q posed at time t as an **instantaneous query** is evaluated as $Q(H_t, t)$

Example:

- "find all motels within 5 kms from my position" → "find all motels within 5 kms distance from the car's position at time *t*"
- Does not mean that only the current database state is used. "Find all motels that I will reach within 10 minutes" refers to all database state with time stamp between now and 10 minutes later.

Continuous Query

Query Q posed at time t as a **continuous query** is evaluated as a sequence of queries

$$Q(H_t, t), Q(H_{t+1}, t+1), Q(H_{t+2}, t+2), \ldots$$

Conceptually, reevaluated on each clock tick as a new instantaneous query.

Example:

- "find all motels within 5 kms from my position" as a continuous query → inform me when suitable motels become available.

Reevaluation on each clock tick not feasible. Instead, we compute a set of tuples with time stamps. Time stamp indicates for which period the tuple belongs to the result.

Continuous query needs to be reevaluated on an explicit update.

FTL - A Query Language Based on Future Temporal Logic

General form of a query:

```
RETRIEVE <target-list> FROM <object-classes> WHERE <FTL-formula>
```

The same form for instantanious and continuous queries.
Mode “continuous query” is set at the user interface level.

Some Example Queries

Example: “Which trucks are within 10 kms of truck T68?”

```
RETRIEVE t
FROM trucks t, trucks s
WHERE s.id = 'T68' ∧ dist(s, t) ≤ 10
```

Example: “Will truck T70 reach its destination within the next 30 minutes?”

```
RETRIEVE t
FROM trucks t
WHERE t.id = 'T70' ∧ eventually_within_30 (dist(t, t.dest) = 0)
```

Example: “Retrieve the friendly helicopters that will arrive in the valley within the next 15 minutes and then stay in the valley for at least 5 minutes”.

```
RETRIEVE h
FROM helicopters h
WHERE eventually_within_15 (inside(h, Valley) ∧
  always_for_5 (inside(h, Valley)))
```

FTL Syntax

Language built from

- symbols
- terms
- formulas

FTL Syntax: Symbols

Definition: The FTL language consists of the following *symbols*:

(i) *Constants*. Example: *10, 'T68', Valley, Time*
(ii) For each $n > 0$, a set of n-ary *function symbols*. Example.: +, *dist*, "."
(iii) For each $n \geq 0$, a set of n-ary *predicate symbols*. Example.: $<$, *inside*
(iv) *Variables*. Example: *s, t*
(v) Logical connectives $\wedge$, $\neg$.
(vi) The *assignment quantifier* $\leftarrow$.
(vii) Temporal modal operators **until**, **nexttime**.
(viii) Brackets and punctuation symbols "(", ")", "[", "]", and ",".

FTL designed to be implemented on top of a DBMS with its own query language. Allows one to embed *atomic queries* from that language.

```
RETRIEVE d.name FROM destinations d WHERE d.id = 'd12'
```

returns a *string* value. Viewed as a *constant symbol*.

```
RETRIEVE d.name FROM destinations d WHERE d.id = y
```

has a free variable *y*. Viewed as a unary *function symbol*.

FTL Syntax: Terms and Formulas

Definition: A *term* is one of the following:

(i) a constant c
(ii) a variable v
(iii) an attribute access $v.A$
(iv) an application of a function to terms of appropriate types $f(t_1, ..., t_n)$.

Some terms: 10, $x + 3$, **dist**(x, y), *d.name,* "retrieve d.name from destinations d where d.id = y".

Definition: A *well-formed formula* is defined as follows:

(i) If R is an n-ary predicate symbol and $t_1, ..., t_n$ are terms of appropriate types, then $R(t_1, ..., t_n)$ is a well-formed formula.
(ii) If f and g are well-formed formulas, then $f \wedge g$ and $\neg f$ are well-formed formulas.
(iii) If f and g are well-formed formulas, then f **until** g and **nexttime** f are well-formed formulas.
(iv) If f is a well formed formula, x is a variable, and t is a term of the same type as x, then $([x \leftarrow t] f)$ is a well formed formula. A variable in a formula is *free*, if it is not in the scope of an assignment quantifier of the form $[x \leftarrow t]$.

FTL Semantics

Meaning of

- symbols
- terms
- formulas

Let s_u the state of the database when the query

```
retrieve <target-list> from <object-classes> where <f>
```

is entered. Semantics of formula f defined with respect to the history H_u.

Symbols

1. *Constants* represent corresponding values from their domain.
    - 54, “T68”, *Valley*, *Time*
2. *Function symbols* have their standard interpretation or denote functions defined in the text.
    - +, *, **dist**
3. *Predicate symbols* also have their standard interpretation.
    - $\leq$

Variable Assignment

Definition: A *variable assignment* for formula f is a mapping μ that assigns to each free variable in f a value from its domain. We denote by $\mu[x/u]$ the mapping obtained from μ by assigning the value u to variable x and leaving all other variables unchanged.

Example:

```
RETRIEVE t
FROM trucks t, trucks s
WHERE s.id = 'T68' ∧ dist(s, t) ≤ 10
```

Assume trucks with identifiers T_1 through T_{100}. Possible assignment:

$$\mu = \{(s, T_{10}), (t, T_{20})\}$$

Terms

Definition: For a term t, its *evaluation* in a state s with respect to a variable assignment μ, denoted $\varphi_{s,\mu}[t]$, is defined as follows:

(i) If t is a constant c, then $\varphi_{s,\mu}[c] = s(c)$.
(ii) If t is a variable v, then $\varphi_{s,\mu}[v] = \mu(v)$.
(iii) If t is an attribute access $v.A$, then $\varphi_{s,\mu}[v.A] = s(\mu[v].A)$.
(iv) If t is an application of function f to arguments $t_1, ..., t_n$, then $\varphi_{s,\mu}[f(t_1, ..., t_n)] = f(\varphi_{s,\mu}[t_1], ..., \varphi_{s,\mu}[t_n])$.

Formulas

Definition: Formula f is *satisfied* at state s on history H with respect to variable assignment μ (satisfied at (s, μ), for short):

(i) $R(t_1, ..., t_n)$ is satisfied $:\Leftrightarrow R(\varphi_{s,\mu}[t_1], ..., \varphi_{s,\mu}[t_n])$ holds.

(ii) $f \wedge g$ is satisfied $:\Leftrightarrow$ both f and g are satisfied at (s, μ).

(iii) $\neg f$ is satisfied $:\Leftrightarrow f$ is not satisfied at (s, μ).

(iv) f **until** g is satisfied $:\Leftrightarrow$ either g is satisfied at (s, μ), or there exists a future state s' on history H such that (g is satisfied at $(s', \mu) \wedge$ for all states s_i on history H before state s', f is satisfied at (s_i, μ)).

(v) **nexttime** f is satisfied $:\Leftrightarrow f$ is satisfied at (s', μ), where s' is the state immediately following s in H.

(vi) $([x \leftarrow t] f)$ is satisfied $:\Leftrightarrow f$ is satisfied at $(s, \mu[x/\varphi_{s,\mu}[t]])$.

Assignment quantifier together with **nexttime** allow one to discover change.

Example: Formula ($[x \leftarrow$ **dist**$(a, b)]$ (**nexttime dist**$(a, b) > x$))

($[x \leftarrow$ **dist**$(a, b)]$ (**nexttime dist**$(a, b) > x$)) is satisfied at state s

$\Leftrightarrow$ (**nexttime dist**$(a, b) > \alpha$) is satisfied at state s, where α is the distance between objects a and b, evaluated in state s

$\Leftrightarrow$ (**dist**$(a, b) > \alpha$) is satisfied at state s'

$\Leftrightarrow$ $(\beta > \alpha)$ where β is the distance between objects a and b, evaluated in state s'

(v) **nexttime** f is satisfied $\Leftrightarrow f$ is satisfied at (s', μ), where s' is the state immediately following s in H.

(vi) $([x \leftarrow t] f)$ is satisfied $:\Leftrightarrow f$ is satisfied at $(s, \mu[x/\varphi_{s,\mu}[t]])$.

Derived Notations

Logical connectives

- $\vee$, $\Rightarrow$

Temporal operators

- **eventually** $g :\Leftrightarrow$ *true* **until** g
- **always** $g :\Leftrightarrow \neg$ (**eventually** $(\neg g)$).

Bounded temporal operators:

- g **until_within_c** $h :\Leftrightarrow[x \leftarrow \mathit{Time}](g$ **until** $(h \wedge \mathit{Time} \leq x + c))$
  - asserts that there exists a future time within at most c time units from now such that h holds, and until then g will be continuously satisfied.
- g **until_after_c** $h :\Leftrightarrow [x \leftarrow \mathit{Time}](g$ **until** $(h \wedge \mathit{Time} \geq x + c))$
  - asserts that there exists a future time after at least c time units from now such that h holds, and until then g will be continuously satisfied.

Other bounded temporal operators:

- **Eventually_within_c** g :⇔ *true* **until_within_c** g
  - formula *g* will be fulfilled within *c* time units from the current state
- **Eventually_after_c** *g* :⇔ *true* **until_after_c** g
  - means that *g* holds after at least *c* units of time
- **Always_for_c** *g* :⇔ *g* **until_after_c** *true*
  - asserts that the formula *g* holds continuously for the next *c* units of time

Finally the temporal operators used in the example queries have been defined.

```
RETRIEVE h
FROM helicopters h
WHERE eventually_within_15 (
 inside(h, Valley) ∧ always_for_5 (inside(h, Valley))
)
```

Evaluating FTL Queries

An algorithm for evaluation. Consider **restricted conjunctive formulas**.

- no negation
- no **nexttime** operator
- no reference to *Time*

But also treat:

- **until_within_c**
- **until_after_c**

Basic idea:

- Formula f with free variables $x_1, ..., x_k$ translated to a relation $R_f(x_1, ..., x_k, t_{start}, t_{end})$.
- For two tuples with the same *instantiation* $\rho = <o_1, ..., o_k>$ of the variables $x_1, ..., x_k$, the time intervals are disjoint and non adjacent.
- A set of tuples T with the same instantiation ρ and set of time intervals I represents a combination of objects $<o_1, ..., o_k>$ that fulfill f during times I.
- Evaluate formula bottom-up, translating connectives into relation operations.

Definition: A *well-formed formula* is defined as follows:

(i) If R is an n-ary predicate symbol and t_1, ..., t_n are terms of appropriate types, then $R(t_1, ..., t_n)$ is a well-formed formula.

(ii) If f and g are well-formed formulas, then $f \wedge g$ [and $\neg f$] are well-formed formulas.

(iii) If f and g are well-formed formulas, then f **until** g [and **nexttime** f] are well-formed formulas.

(iv) ... f **until_within_c** g ...

(v) ... f **until_after_c** g ...

(vi) If f is a well formed formula, x is a variable, and t is a term of the same type as x, then $([x \leftarrow t] f)$ is a well formed formula. A variable in a formula is *free*, if it is not in the scope of an assignment quantifier of the form $[x \leftarrow t]$.

Let h be a subformula with free variables $x_1, ..., x_l$.

Case 1: $h \equiv R(x_1, ..., x_l)$

Example: $h \equiv dist(x_1, x_2) < 8$

Assume for each such atomic predicate an algorithm exists returning for each possible instantiation $<o_i, o_j>$ the time intervals when the predicate holds.

Compute a relation R_h with all such tuples $<o_1, ..., o_l>$ and associated time intervals.

Case 2: $h \equiv f \wedge g$

Let R_f and R_g be the relations computed for the subformulas.

- $R_f(x_1, x_2, x_5, t_s, t_e)$,
- $R_g(x_1, x_4, x_5, x_7, t_s, t_e)$,
- $R_h(x_1, x_2, x_4, x_5, x_7, t_s, t_e)$

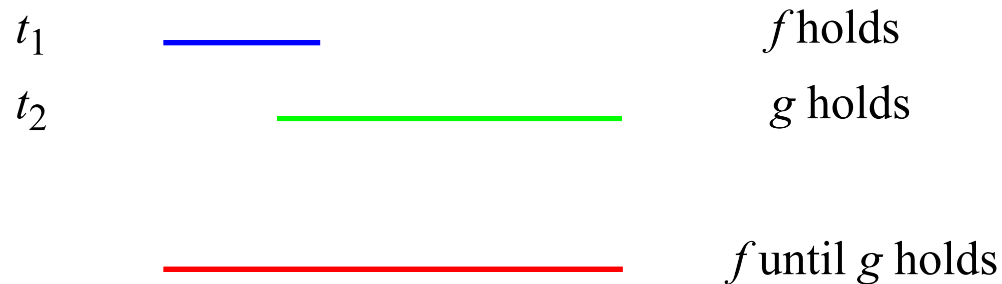
Suppose for an instantiation $<o_1, o_2, o_4, o_5, o_7>$, f is satisfied during I_1 and g is satisfied during I_2. Then $f \wedge g$ is satisfied during $I_1 \cap I_2$.

Compute a join of R_f and R_g; common variables must be equal, and time intervals must intersect. For each result tuple its time interval is the intersection of the time intervals of the two joining tuples.

Case 3: $h \equiv f$ until g

Let R_f and R_g be the relations computed for the subformulas, with p+2 and q+2 attributes.

Consider tuple t_1 in R_f. Let T_1 be the set of all tuples with same values in the first p attributes (same instantiation). Let I_1 be the set of time intervals in T_1. Similarly t_2 in R_g, T_2, I_2.



What about the following situation?

t_1 — *f* holds

t_2 — *g* holds

When does *f* **until** *g* hold?

t_1 — *f* holds

t_2 — *g* holds

f until *g* holds

$chain_1$ $chain_2$

Compute a join of R_f and R_g matching pairs of (sets of) tuples w.r.t. their variable instantiations. For two sets of intervals I_1 and I_2 compute a set of *solution intervals*. Merge overlapping solution intervals to construct *maximal chains.* For each maximal chain, construct one result tuple, with time interval corresponding to the extent of the chain.

Algorithm for merging solution intervals

ASSUMPTIONS:

1. Due to the discrete time model, only closed intervals are considered, i.e., $[t_1, t_2]$.

2. Order on the intervals:

$[t_1, t_2] < [t_3, t_4] \Leftrightarrow t_1 < t_3 \vee (t_1 = t_3 \wedge t_2 < t_4)$.

THE ALGORITHM:

1. Sort the intervals (complexity O(n * log n)).

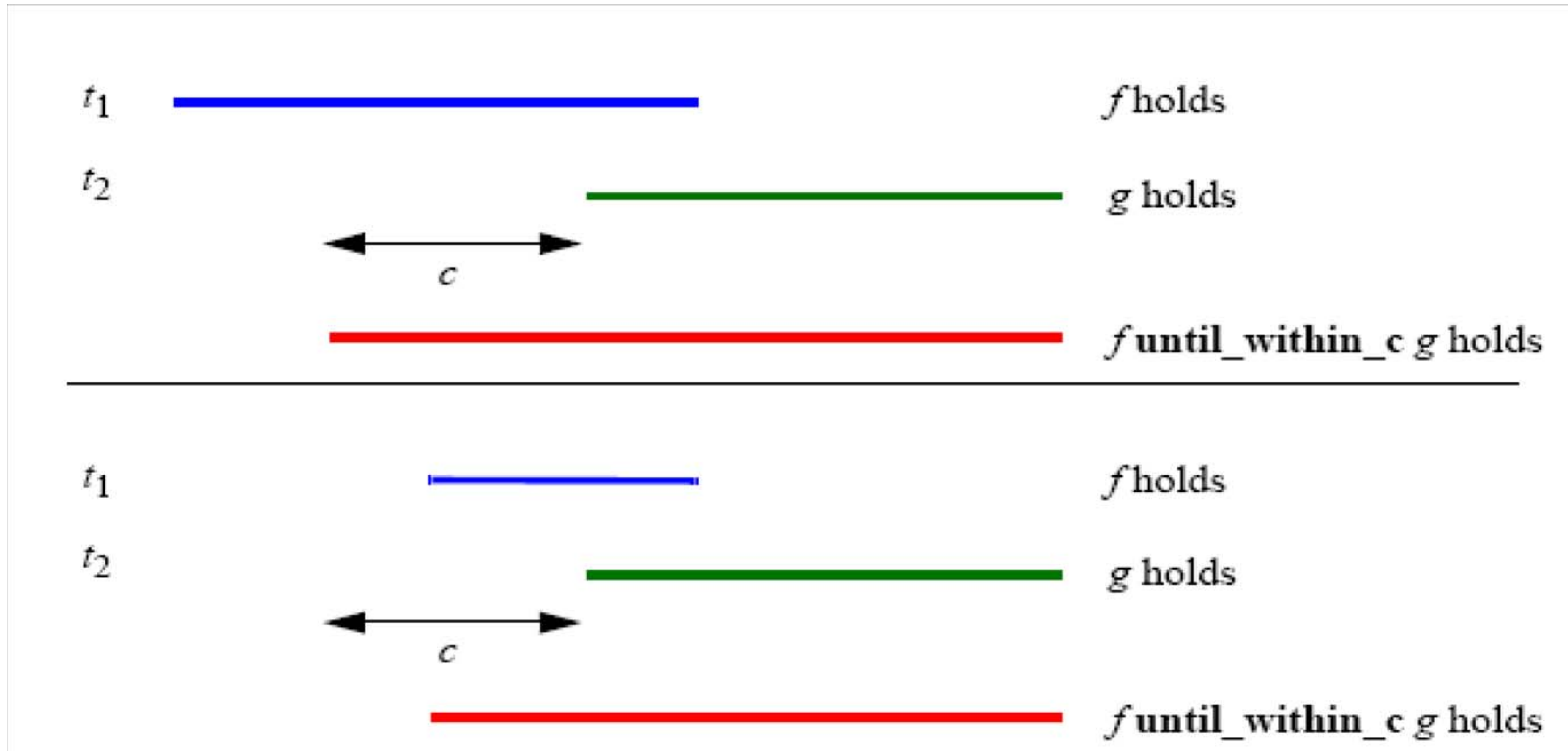
Result: a sequence $<[t_{1,1}, t_{1,2}], [t_{2,1}, t_{2,2}], ..., [t_{n,1}, t_{n,2}]>$.

2. Merge the sequence (complexity O (n)).

TOTAL COMPLEXITY: O(n * log n).

```
result := ◊;
i := 2;
start := t_{1,1};
end := t_{1,2};
while i ≤ n do
  if t_{i,1} > end+1 then
    result := result ∘ ⟨[start, end]⟩;
    start := t_{i,1};
    end := t_{i,2};
    i := i+1
  else if t_{i,2} > end then
    end := t_{i,2};
    i := i+1
  else
    i := i+1
  fi
od
result := result ∘ ⟨[start, end]⟩;
```

Case 4: $h \equiv f$ **until_within_*c*** g



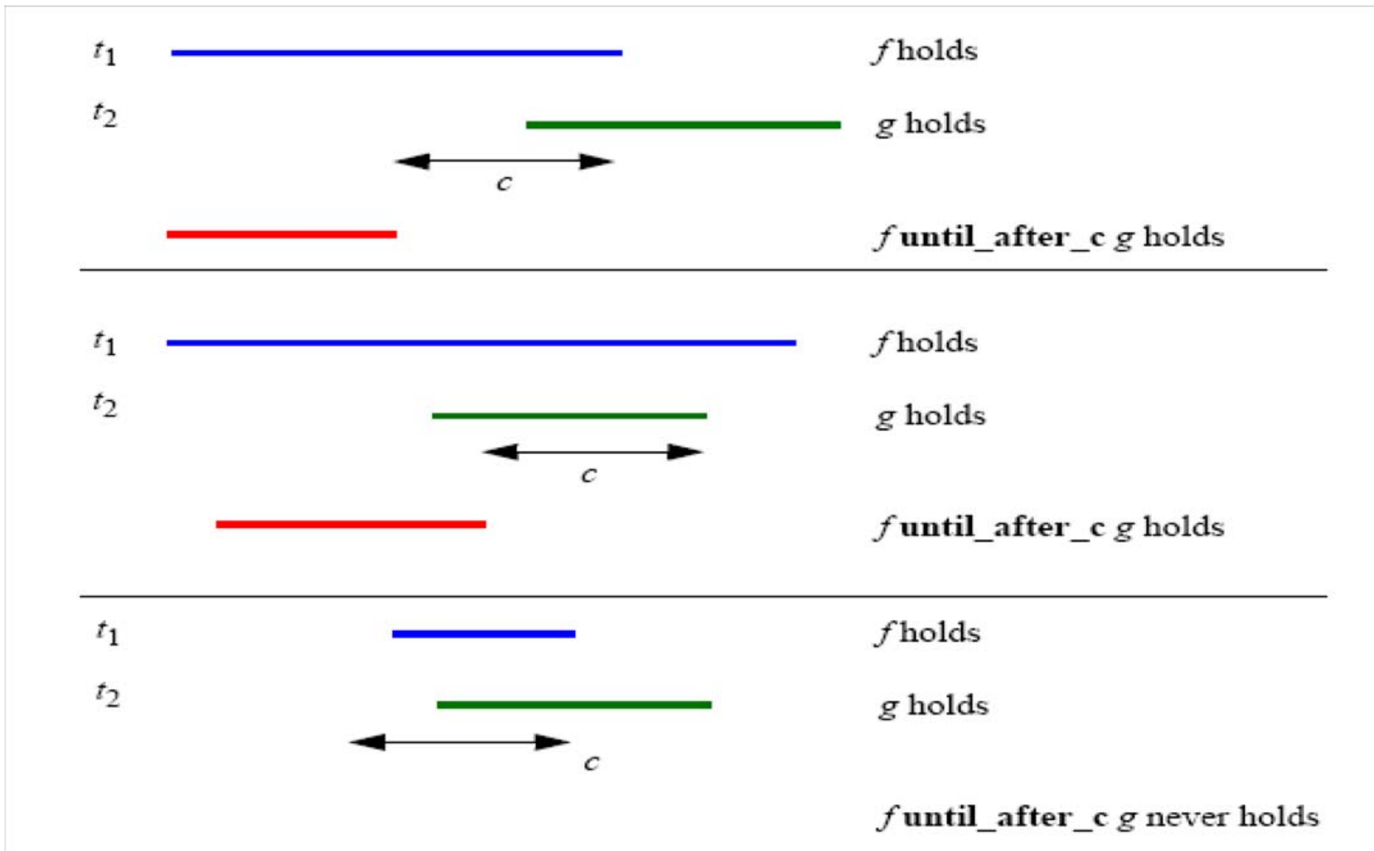
Let t_1, t_2 be matching pairs of tuples from R_f and R_g with overlapping time intervals.

Let $d = \max\{t_1.l,\ t_2.l - c\}$. Then f **until_within_*c*** g holds in the interval $[d,\ t_2.u]$.

Extend to chains as before (i.e., merge overlapping solution intervals).

The result relation is computed in a join, similar to the previous case.

Case 5: $h \equiv f$ until_after_c g



Let t_1, t_2 be matching pairs of tuples from R_f and R_g with overlapping time intervals.

Let $e = \min\{t_1.u, t_2.u\}$. If $t_1.l \leq e - c$, then f **until_within_c** g holds in the interval $[t_1.l, e - c]$.

Extend to chains as before.

Case 6: $h \equiv [y \leftarrow q]\, f$

In general, q contains variables. If there are p variables, the relation for q has $p + 3$ attributes:

- p for variables
- 1 for result
- 2 for time interval

Then compute the join of R_q and R_f on equal variables and overlapping time intervals, as in the other cases. From the result tuples the attribute for variable y is removed (it is no free variable in h).

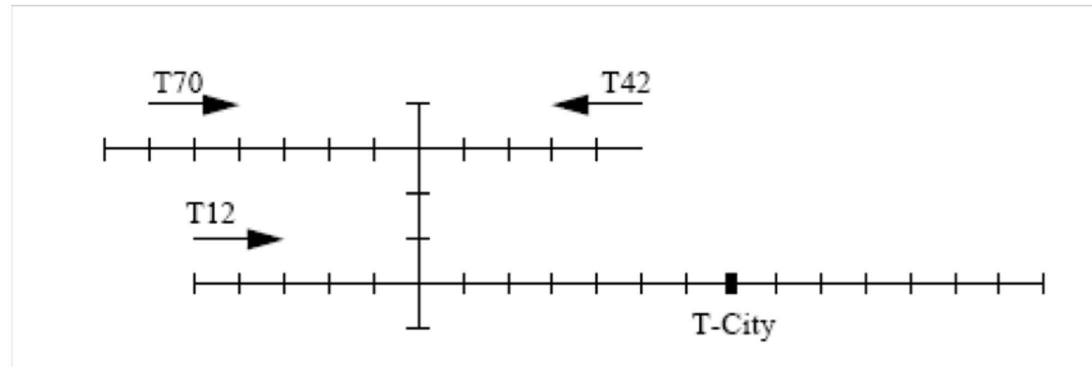
Answering Instantanious and Continuous Queries

Suppose we have the result relation *Answer*(*Q*) .

If Q is an instantanious query,
the instantiations of a tuple belongs to the result
if its time interval contains the current clock tick.

If Q is a continuous query,
at each clock tick, *t*,
the instantiations of a tuple belong to the result
if its time interval contains *t*.

Example of Evaluating FTL Formulas



The unit of the network is 10 kms.
Speed: T12 60 km/h, T42 100 km/h, and T70 120 km/h.

Evaluate the query:

```
RETRIEVE t
FROM trucks t
WHERE eventually_within_30(dist(T, T_City) ≤ 20)
```

Example of Evaluating FTL formulas (cont.)

We rewrite the query:

```
RETRIEVE t
FROM trucks t
WHERE true until_within_30(dist(T, T_City) ≤ 20)
```

First, evaluate the predicate *dist(T, T-City) ≤ 20*.

Truck	*Distance true [km]*	*Distance false [km]*	*Speed [km/h]*	*Minutes needed for 10 kms*
T12	100	140	60	10
T42	130	170	100	6
T70	140	180	120	5

t	*l*	*u*
T12	100	140
T42	78	102
T70	70	90

Second, evaluate the formula *true*.

t	*l*	*u*
T12	0	T_{max}
T42	0	T_{max}
T70	0	T_{max}

Example of Evaluating FTL formulas (cont.)

Third, evaluate *true* **until_within_30** *dist(T, T-City)* $\leq$ *20.*

One tuple in *true* matches one tuple in *dist(T, T-City)* $\leq$ *20* .

t	*l*	*u*
T12	0	T_{max}
T42	0	T_{max}
T70	0	T_{max}

t	*l*	*u*
T12	100	140
T42	78	102
T70	70	90

For each pair of matching tuples, we compute $d = max \{t_1.l, t_2.l - 30\}$.

The result tuple's interval: $[d, t_2.l]$.

t	*l*	*u*
T12	70	140
T42	48	102
T70	40	90

DOMINO prototype system

- Implements MOST and FTL
- Built on top of extensible object-relational DBMSes: Informix (for Unix) and Oracle (for Windows)