Management of integrity constraints
Scuola di Bertinoro
Maggio 1999

Prof., Véronique Benzaken

Université de Paris XI,

L.R.I., UMR 8623 du CNRS

Bat 490, 91405 Orsay Cedex, France

e-mail veronique.benzaken@.lri.fr.

http://www,lri.fr/ benzaken

Contents

Informal presentation of Integrity Constraints.

An integrity constraint is a property that objects stored in a database must satisfy.

Example :

- Each person has an age which ranges between 0 and 130 years.
- Nobody can be married with him(her)self
- Each professor must have been a phd-student in the past
- If a phd-student has got twice zero in the past then he must be fired

Informal presentation of Integrity Constraints.

The first two examples are refered as static integrity constraints while the last ones are called temporal (or dynamic) integrity constraints.

- Static integrity constraints are stated on a single database state.
- Temporal integrity constraints are stated on a sequence of database states.

Relational theory of dependencies

Background : In the relational data model, relations are represented by a (labeled) cartesian-product of elementary domains.

Movie(title, director, actor, time, color, producer)

- title, director, actor, time, color, producer, are called attributes in the relational terminology

- title, director, actor, producer range in the string domain

- time, color in the float and boolean domains respectively

Relational theory of dependencies

Dependencies are special cases of static integrity constraints.

- Functional dependencies
- Multi-valued dependencies
- Inclusion dependencies

Functional dependencies

Assume that we want to assert that every database instance that represents movies is such that given a producer and a title there is only one director.

Relational model expression for this is:

$$\text{producer, title} \rightarrow \text{director}$$

which is called a Functional dependency.

Given a relation $R(T)$ with $X, Y \subseteq T$,

$$X \rightarrow Y \Leftrightarrow \forall\, t1,\, t_2 \in R,\; t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$$

Multi-valued dependencies

Given a relation R(T) with X, Y $\subseteq$ T,

$$X \rightarrow\rightarrow Y \Leftrightarrow \forall\ t1,\ t_2 \in R,\ \text{if } t_1[X] = t_2[X] \Rightarrow$$
$$\exists\ t_3 \text{ and } t_4 \in R \text{ such that}$$

- $t_1[X] = t_2[X] = t_3[X] = t_4[X]$
- $t_3[Y] = t_1[Y]$ and $t_3[T\text{-}X\text{-}Y] = t_2[T\text{-}X\text{-}Y]$ and
- $t_4[Y] = t_2[Y]$ and $t_4[T\text{-}X\text{-}Y] = t_1[T\text{-}X\text{-}Y]$

We can exchange the Y values of t1 and t_2 to obtain two new tuples that must also be in R.

Inclusion dependencies

Given relations R(T) and S(U) and attributes A_1, ..., $A_n \in$ T B_1, ..., $B_m \in$ U, there is an inclusion dependency between R and S on A_i and B_j if:

$$R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_m]$$

Management of integrity constraints

Database programming languages must provide the following mechanisms:

- A language for expressing integrity constraints

- A way of checking such constraints.
  - Given a (sequence of) database state(s) satisfying the (possibly temporal) constraints, establish whether the new state resulting of the execution of a given program (called a transaction) still satisfies the constraint.
  - If such is not the case, the transaction effect must be cancelled.

Management of integrity constraints

Expression of integrity constraints

- Must be expressed at the schema level
- As far as possible in a global and declarative way.

Checking of constraint

- Dynamic checking
- Static checking

A bunch of integrity constraints

- There must be at least one Person who is not married.
- A person cannot be married to itself.
- No parent can be married to one of its children.
- An employee who is not married must have a boss.
- A person must have some money.
- A parent who is not married cannot have any children.
- There must be at least one Person who has no best friend.
- If a PhD-student is advised by Prof B then sometimes in the past he (or she) has been a student of Prof B.
- If a Person is married then sometimes in the past he (or she) has been single.
- If a PhD-student has obtained at least twice the note 0, then he (or she) has no longer any advisor.

Expression of integrity constraints: Galileo97

```
let rec Persons class Person <-> [
    name        :  string,
    spouse      :  var Person,
    bestfriend  :  var Person,
    money       :  var int
    marry       := meth(q: Person) : null is
                    if (self!=q) then (
                     self.spouse <- q;
                     q.spouse <- self )
                   else failwith ‘‘Nobody can be
                                married with itself’’
    separate    := meth() : null is
                    self.spouse <- nil
    spend       := meth(a : int) : null is
                    if ((at self.money - a) > 0) then
                     at self.money <- at self.money - a
                    else failwith ‘‘amount must be > 0’’
    setbestf    := meth(q: Person) : null is
                    if (some u in Persons
                     with (u !=self And u.bestfriend = nil))
                    then self.bestfriend <- q
                    else failwith ‘‘somebody must have
                                 no bestfriend’’]
    before mk(p)
     if (at p.money < 0) do failwith ‘‘amount of money
                                       must be > 0’’
     if (p.spouse = p)
       do failwith ‘‘nobody can be married with itself’’
```

Expression of integrity constraints: Galileo97

```
let rec Parents subset of Persons class
    Parent <-> is Person and
    [children: var seq (var Person)]
    before mk(p)
    if !(p.spouse != nil  Or p.children = seq())
         do failwith ‘‘No children if not married’’
    if !(p.spouse = nil  Or p.spouse notin p.children)
         do failwith ‘‘No parent can be married
                       with one of its child’’

let rec  Employees subset of Persons class
    Employee <-> is Person and
    [boss: var Person]
    before mk(e)
    if !(e.spouse != nil Or e.boss != nil)
         do failwith ‘‘If not married
                       one must have a boss’’
```

• • •

Expression of integrity constraints: O_2

```
class Person inherit Object public type tuple(
          name: string,
          spouse: Person,
          bestfriend: Person,
          money: integer )
   method marry(q: Person);
          separate();
          spend(amount: integer);
          setbestfriend(p: Person);
end;

class Parent inherit Person public type tuple(
          children: set (Person))
   method marry(q: Person);
          separate();
          spend();
end;

class Employee inherit Person  public type tuple(
          boss: Person)
   method separate();
end;
```

Expression of integrity constraints: O_2

```
class Prof inherit Person type tuple(
        student: set(Person))
end;

class PhD-student inherit Person type tuple(
        advisor: Prof,
        note:integer)
        setadvisor(p:Prof);
        fire;
end;

name Persons: set (Person);
name Parents: set (Parent);
name Employees: set (Employee);
name Profs: set(Prof);
name PhD-students: set (PhD-student);
```

Expression of integrity constraints: O_2

```
constraints

C1:  exists x in Persons:  x.spouse==nil;
C2:  forall x in Persons:  x.spouse!=x;
C3:  forall x in Parents:  !(x.spouse in x.children);
C4:  forall x in Employees:  x.spouse==nil
==> x.boss!=nil;
C5:  forall x in Persons:  x.money > 0;
C6:  forall x in Parents:  x.spouse==nil ==>
       !(exists y in x.children);
C7:  exists x in Persons:  x.bestfriend==nil;
C8:  forall x in PhD-students:  x.advisor==y
```

==>$\overleftarrow{\Diamond}$ `(x in y.student)`

`C9:  forall x in Persons:`

`x.spouse`$\neq$`nil`$\rightarrow \overleftarrow{\Diamond}$ `(x.spouse == nil)`

`C10:  forall x in PhD-students:`

$\overleftarrow{\Diamond}$`(x.note==0` $\wedge \overleftarrow{\Diamond}$ `x.note==0) ==> x.advisor==nil`

```
end;
```

Expression of integrity constraints: O_2

```
method separate()
in class Person {
   self.spouse:=nil;}
```

```
method spend(a:integer)
in class Person {
   if (self.money-a>0)
     self.money:=self.money-a;}
```

```
method setbestfriend(q: Person)
in class Person {
   if (exists u in Persons:
       u!=self && u.bestfriend=nil)
         self.bestfriend:=q;}
```

Expression of integrity constraints: O_2

method marry(q:Person) in class Person {
if ($\overleftarrow{\Diamond}$ self.spouse = nil) && (self!=q
self.spouse:=q;}

method setadvisor(b:Prof) in class PhD-student{
if ($\overleftarrow{\Diamond}$ self $\in$ b.student)
self.advisor:=b;}

method fire in class PhD-student{
if ($\overleftarrow{\Diamond}$(self.note=0 $\wedge\overleftarrow{\Diamond}$self.note=0))
advisor:=nil;}

Expression of integrity constraints: O_2

```
method separate()
in class Employee {
  if (self.boss!=nil)
    self.spouse:=nil;}
```

```
method separate()
in class Parent {
 if (!(exist u in self.children))
    self.spouse:=nil;}
```

```
method marry(q:Person)
in class Parent {
  if (self != q
      && !(q in self.children)
      && !(self in q.children))
       {self.spouse:=q;
        q.spouse:=self;}}
```

```
method spend (a:integer)
in class Parent {
if (self.money-a>0)
 self.money=self.money-a;
forall u
where (u in self.children)
do {if (u.money-a>0)
    u.money=u.money-a;}}
```

Pros and Cons of Triggers

Pros

- The application is relieved from controls that are operated by triggers.
- Defined and managed in a centralized way: users cannot bypass such controls.

Cons

- Different semantics according to the system
  - Granularity
  - Conflict resolution
  - Cascade Triggers

Pros and Cons of Triggers

Cons

- Triggers and Transactions
- Complexity
- Rigidity

- With respect to integrity: knowledge spread out among triggers

Dynamic Checking of Constraints

Transactions are atomic, concurrent, isolated and durable.

Delicate phase: commit.

Not only, the DBMS have to check whether everything performed correctly w.r.t, concurency but it also have to check that the new state produced still satisfies the constraints.

Therefore additional run-time tests are (automatically) generated by the compiler. Those tests may be redundant w.r.t., precautions already taken by the programmer.

Problem

- Some methods can never violate some integrity constraints.

- Likely cause: defensive programming. Run time tests encoded in the body of the methods.

- Defensive programming must be detected at compile-time:

  - to avoid the run-time overhead of integrity checking.

  - to warn the programmer.

- Fact: defensive programming is commonly used in real systems.

Static checking of integrity constraints

Problem

- As far as possible: avoid dynamic checking of constraints.
- Prove at compile time that updates are safe (if such is the case)

A solution

- Capture (partially) the semantics of updates
- Use a instance of abstract interpretation.

Case of Static Integrity Constraints

Forward and Backward Predicate Transformers

Function.

Input: formula ϕ and method m.

Output: formula $\mathcal{POST}(m, \phi)$ or $\mathcal{PRE}(m, \phi)$.

Correctness requirement (to be proven):

$$
\begin{tikzcd}
\mathcal{PRE}(m,\phi) \arrow[r, "\mathcal{PRE}(m)"] & \phi \\
\mathcal{B}_{\texttt{old}} \arrow[u, dash, "\models"] \arrow[r, "m"] & \mathcal{B}_{\texttt{new}} \arrow[u, dash, "\models"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\phi \arrow[r, "\mathcal{POST}(m)"] & \mathcal{POST}(m,\phi) \\
\mathcal{B}_{\texttt{old}} \arrow[u, dash, "\models"] \arrow[r, "m"] & \mathcal{B}_{\texttt{new}} \arrow[u, dash, "\models"']
\end{tikzcd}
$$

Backward *Forward*

m preserves C if $\mathcal{POST}(m, C) \Rightarrow C$.

m preserves C if $C \Rightarrow \mathcal{PRE}(m, C)$.

Forward and backward predicate transformers

- Preliminary translation into intermediate language.

- Avoid navigation in assignments and formulae.

- New instruction:

  forone *object* **where** *condition* **do** *instruction*

- Example:

  p.spouse.spouse:=q;

  becomes

  forone x **where** x=p.spouse **do** x.spouse:=q;

Forward and backward predicate transformers

- Similar decomposition of formulae.

- A formula ϕ is decomposed if it only contains atoms of the form:
  - $R(x_1, x_2)$, where R is a binary relation symbol ($=$,$\neq$,$>$, ...).
  - $R(f(x_1, x_2), x_3)$, where R is either "$=$", or "$\neq$", and f a function symbol
  - x_1, x_2 and x_3 denote variables or constants.

- Example

```
forall x :  x.a > 3
```

is decomposed in

```
forall x, exists y :  x.a = y ∧ y > 3
```

Forward and Backward Predicate Transformers

Method = program

$m(p_1, ..., p_n)$ =*instruction*

m : *name* of the method, p_1,...,p_n : parameters, *instruction*: *body* of m.

Syntax:

$$
\begin{array}{lll}
\textit{instruction} & ::= & \textit{parameter.attribute} := \textit{parameter} \\
& | & \textit{instruction} \ ; \ \textit{instruction} \\
& | & \{ \ \textit{instruction} \ \} \\
& | & \texttt{if} \ \textit{condition} \ \texttt{then} \ \textit{instruction} \\
& | & \texttt{forone} \ \textit{parameter} \ \texttt{where} \ \textit{condition} \ \texttt{do} \ \textit{instruction} \\
& | & \texttt{forall} \ \textit{parameter} \ \texttt{where} \ \textit{condition} \ \texttt{do} \ \textit{instruction}
\end{array}
$$

A Forward Predicate Transformer for Static Constraints

Formula parse:

- $\mathcal{POST}(m, \phi \land \psi) \equiv \mathcal{POST}(m, \phi) \land \mathcal{POST}(m, \psi)$
- $\mathcal{POST}(m, \phi \lor \psi) \equiv \mathcal{POST}(m, \phi) \lor \mathcal{POST}(m, \psi)$
- $\mathcal{POST}(m, \texttt{forall } x : \ \phi(x)) \equiv \texttt{forall } x : \ \mathcal{POST}(m, \phi(x))$
- $\mathcal{POST}(m, \texttt{exists } x : \ \phi(x)) \equiv \texttt{exists } x : \ \mathcal{POST}(m, \phi(x))$

A Forward Predicate Transformer for Static Constraints

Method parse:

- If: $m \;\equiv\; u.a{:=}v$ and ϕ is a literal, then :
  - If $\phi \equiv (x.a = y)$ then
    $\mathcal{POST}(m, \phi) \equiv (u = x \wedge u.a = v) \vee (u \neq x \wedge u.a = v \wedge x.a = y)$
  - If $\phi \equiv (x.a \neq y)$ then
    $\mathcal{POST}(m, \phi) \equiv (u = x \wedge u.a = v) \vee (u \neq x \wedge u.a = v \wedge x.a \neq y)$
  - Else $\mathcal{POST}(m, \phi) \equiv \phi \wedge u.a = v$

A Forward Predicate Transformer

- If $m \equiv i_1;i_2$ then
  $\mathcal{POST}(m, \phi) \equiv \mathcal{POST}(i_2, \mathcal{POST}(i_1, \phi))$

- If $m \equiv \{i\}$ then
  $\mathcal{POST}(m, \phi) \equiv \mathcal{POST}(i, \phi)$

- If $m \equiv \texttt{if}\ \psi\ \texttt{then}\ i$ then
  $\mathcal{POST}(m, \phi) \equiv \mathcal{POST}(i, \psi \wedge \phi) \vee (\neg\psi \wedge \phi)$

- If $m \equiv \texttt{for one}\ v\ \texttt{where}\ \psi(v)\ \texttt{do}\ i$ then
  $\mathcal{POST}(m, \phi) \equiv \texttt{exists}\ v : \mathcal{POST}(i, \psi(v) \wedge \phi)$

Loops

Intuition :

Attempt to prove constraint C as a loop invariant

The forward analysis of loop l:

`forall` *variable* `where` *condition* `do` *instruction*

is:

$$\mathcal{POST}(l, \phi) = \begin{cases} C & \textbf{if } \phi \Rightarrow C \textbf{ and } \mathcal{POST}(i, C) \Rightarrow C \\ \texttt{true} & \textbf{otherwise.} \end{cases}$$

We replace ϕ by C if ϕ implies C and if instructions contained within the loop preserve C,

Else we replace it by `true`.

A Backward Predicate Transformer

Method parse:

- If $m \equiv$ `u.a:=v` and ϕ is a literal then
  - If $\phi \equiv (x.a = y)$ $\mathcal{PRE}m(\phi) \equiv$ $(u = x \wedge v = y) \vee (u \neq x \wedge x.a = y)$
  - If $\phi \equiv (x.a \neq y)$ $\mathcal{PRE}(m, \phi) \equiv$ $(u = x \wedge v \neq y) \vee (u \neq x \wedge x.a \neq y)$
  - Else $\mathcal{PRE}(m, \phi) \equiv \phi$

- If $m \equiv i_1$;i_2 then $\mathcal{PRE}(m, \phi) \equiv \mathcal{PRE}(i_1, \mathcal{PRE}(i_2, \phi))$
- If $m \equiv \{i\}$ then $\mathcal{PRE}(m, \phi) \equiv \mathcal{PRE}(i, \phi)$
- If $m \equiv$ `if` ψ `then` i then $\mathcal{PRE}(m, \phi) \equiv$ $(\psi \wedge \mathcal{PRE}(i, \phi)) \vee (\neg\psi \wedge \phi)$
- If $m \equiv$ `for one` v `where` $\psi(v)$ `do` i then $\mathcal{PRE}(m, \phi) \equiv$ `exists` $v : \psi(v) \wedge \mathcal{PRE}(i, \phi)$

Soundness and Application

Theorem 1 Let $\mathcal{B}_{\texttt{old}}$ be a database, m be a method, and $\mathcal{B}_{\texttt{new}}$ be the database before the execution of m on $\mathcal{B}_{\texttt{old}}$. Let ϕ be a formula. Then:

$$\mathcal{B}_{\texttt{old}} \models \phi \implies \mathcal{B}_{\texttt{new}} \models \mathcal{POST}(m, \phi)$$

$$\mathcal{B}_{\texttt{old}} \models \mathcal{PRE}(m, \phi) \implies \mathcal{B}_{\texttt{new}} \models \phi$$

Corollary 1 Let C be an integrity constraint, and m be a method. If $\mathcal{POST}(m, C) \Rightarrow C$ then m preserves C. If $C \Rightarrow \mathcal{PRE}(m, C)$ then m preserves C.

Use of two theorem provers. Forward: Tableaux-based, Backward: Resolution-based.

Corollary 2 Let $m = \{m_1, ..., m_n\}$ an overloaded method, and C be an integrity constraint. If $\forall i \in [1..n], \mathcal{POST}(m_i, C) \Rightarrow C$ or $C \Rightarrow \mathcal{PRE}(m_i, C)$ then m preserves C.

Approach ensures modularity.

Experiment

Sun sparc (sun4m) server Solaris 2.3 OS

method	class	constraint	forward	backward
separate	Person	C1	0.21s.	0.12s.
separate	Employee	C1	0.27s.	0.23s.
separate	Parent	C1	0.29s.	0.57s.
separate	Parent	C6	1.21s.	2.48s.
separate	Employee	C4	1.06s.	0.22s.
marry	Person	C2	0.41s.	0.33s.
marry	Parent	C2	1.30s.	9.04s
marry	Parent	C3	3.60s.	1h.06min.35s.
spend	Person	C5	3.68s.	76.42s.
spend	Parent	C5	1.71s.	153.04s
setbestfriend	Person	C7	3.26s.	0.31s.

Experiment

```
method separate() in class Person {
   self.spouse:=nil;}
```

```
C1: exists x in Persons: x.spouse==nil;
```

Time: forward 0.21s. backward 0.12s.

```
method separate() in class Employee{
  if (self.boss!=nil)
      self.spouse:=nil;}
```

```
C1: exists x in Persons: x.spouse==nil;
```

Time: forward 0.27s. backward 0.23s.

```
C4: forall x in Employees:
     x.spouse==nil ==> x.boss!=nil;
```

Time: forward 1.06s. backward 0.22s.

Experiment

```
method separate in class Parent{
       if (!(exist u in self.children))
          self.spouse:=nil;}
```

```
C1: exists x in Persons: x.spouse==nil;
```

Time: forward 0.29s. backward 0.57s.

```
C6: forall x in Parents:
     x.spouse==nil ==>!(exists y in x.children)
```

Time: forward 1.21s. backward 2.48s.

Experiment

```
method marry(q:Person) in class Person{
  if (self!=q){
    self.spouse:=q;
    q.spouse:=self;}}
```

```
C2: forall x in Persons: x.spouse!=x;
```

Time: forward 0.41s. backward 0.33s.

Experiment

```
method marry(q:Person) in class Parent{
  if (self != q
      && !(q in self.children)
      && !(self in q.children)){
     self.spouse:=q;
     q.spouse:=self;}}
```

```
C2: forall x in Persons: x.spouse!=x;
```

Time: forward 1.30s. backward 9.04s

```
C3: forall x in Parents:
     !(x.spouse in x.children);
```

Time: forward 3.60s. backward 1h.06min.35s.

Experiment

```
method spend(a:integer) in class Person {
        if (self.money-a>0)
          self.money:=self.money-a;}
```

```
C5: forall x in Persons: x.money > 0;
```

Time: forward 1.71s.. backward 76.42s.

Experiment

```
method spend (a:integer) in class Parent
{
  if (self.money-a>0)
        self.money=self.money-a;
  forall u where (u in self.children) do
   {
    if (u.money-a>0)
        u.money=u.money-a;
   }
}
```

```
C5: forall x in Persons: x.money > 0;
```

Time: forward 3.64s backward 153.04s

Correcting unsafe updates

- $\mathcal{PRE}$ can be used in order to repair unsafe updates.

- However, it remains unclear that the performance yielded will be reasonable.

Case of Dynamic Integrity Constraints

Dynamic integrity constraints management

The formal language is the language of first-order *Linear Temporal Logic* (LTL) (with equality).

We consider a version with *finite* time, noted LTL^{fin} (standard in databases).

The language extends classical first order predicate calculus, by means of the unary past time operators

- $\overleftarrow{\Box}$ (always in the past) and
- $\overleftarrow{\Diamond}$ (sometimes in the past).

Limit ourselves to "backwards" reasoning: Only the time interval going from the current time point back to the initial one may be observed.

Syntax and Semantics of LTL^{fin}

- A set of predicate symbols P (contains, in particular, the binary predicate $=$, whose fixed interpretation is equality.)

- A set of function symbols $F = F_{rigid} \cup F_{non-rigid}$.

A temporal interpretation is a finite sequence of classical interpretations $\mathcal{M}_0$, $\mathcal{M}_1$,..., $\mathcal{M}_k$, where 0,1,...,k are time points.

Syntax and Semantics of LTL^{fin}

A temporal frame T is an initial segment of the natural numbers: $\langle 0, \cdots, k \rangle$; its elements are called *time points*.
A temporal interpretation $\mathcal{M}$ is a 5-tuple

$$\langle T, D, \delta_P, \delta_{Fr}, \delta_{Fnr} \rangle$$

where:

- T is temporal frame;
- D is a non empty set (the domain, possibly infinite)

Syntax and Semantics of LTL^{fin}

- δ_P is a mapping from $P \times T$ to the set of relations over D such that, for each n-ary predicate symbol p, where $n > 0$, and for each $i \in T$, $\delta_P(p, i)$ is a subset of D^n;
- δ_{Fr} is a mapping from F_{rigid} to the set of (total) functions on D such that, for any n-ary function symbol f with $n \geq 0$, $\delta_{Fr}(f)$ is a function from D^n to D;
- δ_{Fnr} is a mapping from $F_{non-rigid} \times T$ to the set of (total) functions on D such that, for any n-ary function symbol f with $n \geq 0$, for any $i \in T$, $\delta_{Fnr}(f, i)$ is a function from D^n to D.

Syntax and Semantics of LTL^{fin}

- Variable assignment σ:Var $\rightarrow D$ and

- A x-variant of σ is a variable assignment σ' such that, for any variable y other than x, $\sigma(y) = \sigma'(y)$.

- Given any term t, and any time point i, the denotation of t w.r.t. the interpretation $\mathcal{M}$, the variable assignment σ and the time point i, is noted $[t]_{\mathcal{M},\sigma,i}$

Syntax and Semantics of LTL^{fin}

1. If t is a variable, then $[t]_{\mathcal{M},\sigma,i} = \sigma(x)$.

2. If t is $f(t_1, ..., n)$, where $f \in F_{rigid}$, then $[t]_{\mathcal{M},\sigma,i} = \delta_{Fr}(f)([t_1]_{\mathcal{M},\sigma,i}, ..., [t_n]_{\mathcal{M},\sigma,i})$.

3. If t is $f(t_1, ..., n)$, where $f \in F_{non-rigid}$, then $[t]_{\mathcal{M},\sigma,i} = \delta_{Fnr}(f, i)([t_1]_{\mathcal{M},\sigma,i}, ..., [t_n]_{\mathcal{M},\sigma,i})$.

Syntax and Semantics of LTL^{fin}

Given any formula F, F is satisfied by $\mathcal{M}$ w.r.t. the variable assignment σ at the time point i noted by: $\mathcal{M}_{\sigma,i} \models F$

1. $\mathcal{M}_{\sigma,i} \models \top$ (i.e. $\top$ denotes the boolean "True")
2. $\mathcal{M}_{\sigma,i} \models p(t_1,..,t_n)$ iff $\langle [t_1]_{M,\sigma,i}, .., [t_n]_{M,\sigma,i}\rangle \in \delta_P(p)$, for $p \in P$.
3. $\mathcal{M}_{\sigma,i} \models t_1 = t_2$ iff $[t_1]_{M,\sigma,i} = [t_2]_{M,\sigma,i}$
4. $\mathcal{M}_{\sigma,i} \models \neg A$ iff $\mathcal{M}_{\sigma,i} \not\models A$
5. $\mathcal{M}_{\sigma,i} \models A \wedge B$ iff $\mathcal{M}_{\sigma,i} \models A$ and $\mathcal{M}_{\sigma,i} \models B$.
6. $\mathcal{M}_{\sigma,i} \models A \vee B$ iff either $\mathcal{M}_{\sigma,i} \models A$ or $\mathcal{M}_{\sigma,i} \models B$.
7. $\mathcal{M}_{\sigma,i} \models \forall x A(x)$ iff for any x-variant σ' of σ, $\mathcal{M}_{\sigma',i} \models A(x)$
8. $\mathcal{M}_{\sigma,i} \models \exists x A(x)$ iff there exists a x-variant σ' of σ such that $\mathcal{M}_{\sigma',i} \models A(x)$
9. $\mathcal{M}_{\sigma,i} \models \overleftarrow{\Box} A$ iff for all $j < i$, $\mathcal{M}_{\sigma,j} \models A$.
10. $\mathcal{M}_{\sigma,i} \models \overleftarrow{\Diamond} A$ iff there exists a $j \in T$ such that $j < i$ and $\mathcal{M}_{\sigma,j} \models A$.

Syntax and Semantics of LTL^{fin}

Note that the semantics for the operators $\overleftarrow{\Box}$ and $\overleftarrow{\Diamond}$ is "strict".

Thus, $\overleftarrow{\Box}A$ ($\overleftarrow{\Diamond}A$) holds at time point i iff A holds at all (some) time points strictly preceeding i.

The operators corresponding to the "large" semantics of $\overleftarrow{\Box}$ and $\overleftarrow{\Diamond}$, are noted, respectively, $\overleftarrow{\Box}_l$ and $\overleftarrow{\Diamond}_l$ and are defined by:

$$\overleftarrow{\Box}_l A =_{def} A \wedge \overleftarrow{\Box} A \qquad\qquad \overleftarrow{\Diamond}_l A =_{def} A \vee \overleftarrow{\Diamond} A$$

Syntax and Semantics of LTL^{fin}

Definition 1 [Validity] A closed formula A is *valid* iff A holds in any time point of any temporal interpretation $\mathcal{M}$.

We also make use of a weaker notion

Definition 2 [k-Validity] Let k be any natural number. A closed formula A is k-*valid* iff A holds in any time point of any temporal interpretation $\mathcal{M}$ having a temporal frame $T = \langle 0, ..., m \rangle$ where $m \leq k$.

Modeling evolving databases

The unique domain D models the set of the (potential) object identifiers in the database.

Let B be an instance of a database evolving through time and let $B_0, \cdots, B_n$ be its *history* up to current time n,

We identify any base B_i with a classical interpretation

The database history $B_0, \cdots, B_n$ the temporal interpretation $\mathcal{M}_B$ with

$$\langle 0, ..., n \rangle$$

as temporal frame and, for any closed formula A, for any $i \in \langle 0, \cdots, n \rangle$, $\mathcal{M}_{Bi} \models A$ if and only if A is true w.r.t. the base B_i.

Given any closed LTL^{fin} formula A the history $B_0, \cdots, B_n$ satisfies A if and only if, for each i such that $0 \leq i \leq n$, $\mathcal{M}_{Bi} \models A$.

Modeling evolving databases

In our approach, a dynamic constraint is expressed by a closed LTL^{fin} formula. For instance, a constraint of the form:

If A is true now then B has been sometimes true in the past

is formalized as

$$\forall \overrightarrow{x}(A(\overrightarrow{x}) \rightarrow \overleftarrow{\Diamond}_* B(\overrightarrow{x}))$$

and a constraint of the form:

If A is true now then B has always been true in the past

is formalized as:

$$\forall \overrightarrow{x}(A(\overrightarrow{x}) \rightarrow \overleftarrow{\Box}_* B(\overrightarrow{x}))$$

where $\overrightarrow{x}$ denotes a variable vector $x_1, ..., x_n$ and $\overleftarrow{\Diamond}_*$ (resp. $\overleftarrow{\Box}_*$) represents $\overleftarrow{\Diamond}$ or $\overleftarrow{\Diamond}_l$ (resp. $\overleftarrow{\Box}$ or $\overleftarrow{\Box}_l$).

$\mathcal{POST}$ for temporal constraints

The predicate transformer $\mathcal{POST}$ for temporal constraints expressed by a LTL^{fin} formula ϕ (in nnf) is extended to the temporal connectives as follows:

- Formula parsing:
  1. $\mathcal{POST}(m, (\overleftarrow{\Diamond}\phi)) \equiv \overleftarrow{\Diamond}\phi$
  2. $\mathcal{POST}(m, (\overleftarrow{\Box}\phi)) \equiv \overleftarrow{\Box}\overleftarrow{\Box}\phi$
- Method's body parsing does not change.

$\mathcal{PRE}$ for temporal constraints

- Formula parsing:
  1. $\mathcal{PRE}(m, (\overleftarrow{\Diamond}\phi)) \equiv \overleftarrow{\Diamond}_l\, \phi$
  2. $\mathcal{PRE}(m, \overleftarrow{\Box}\phi) \equiv \overleftarrow{\Box}_l \phi$

- Method's body parsing does not change.

Soundness of $\mathcal{POST}$ and $\mathcal{PRE}$

Theorem 2 Let $B = B_0, \cdots, B_{old}$ be a database history, ϕ any LTL^{fin} formula and m any method. Let B_{new} be the base resulting from the execution of m on B_{old}.

- If $B_{old} \models \phi$ then $B_{new} \models \mathcal{POST}(m, \phi)$.
- If $B_{old} \models \mathcal{PRE}(m, \phi)$ then $B_{new} \models \phi$

Safe Methods

Definition 3 A method m is safe w.r.t. an integrity constraint ϕ if, whenever $B_0, \cdots, B_n$, where $0 \leq n$, is a database history satisfying ϕ and B_{n+1} is obtained by executing m on B_n, the new history $B_0, \cdots, B_n, B_{n+1}$ satisfies ϕ.

Corollary 3 Let m be a method and ϕ a constraint. Method m is safe w.r.t. ϕ if:
(1) $\mathcal{POST}(m, \phi) \Rightarrow \phi$ is valid
or if
(2) $\phi \Rightarrow \mathcal{PRE}(m, \phi)$ is valid.

k-safe Methods

Besides the notion of safe method, we need a weaker notion.

Definition 4 Let k be a positive integer. A method m is k-safe w.r.t a constraint ϕ if, whenever $B_0, \cdots, B_n$ is a database history satisfying ϕ, $0 \leq n < k$ and B_{n+1} is obtained by executing m on B_n, the new history $B_0, \cdots, B_n, B_{n+1}$ satisfies ϕ.

Corollary 4 Let m be a method and ϕ a constraint. Method m is k-safe w.r.t. constraint ϕ if:
(1) $\mathcal{POST}(m, \phi) \Rightarrow \phi$ is k-valid
or if
(2) $\phi \Rightarrow \mathcal{PRE}(m, \phi)$ is k-valid

Some Remarks

Due to the intrinsic incompleteness of the logic LTL^{fin}

The weaker notion of k-validity, allowing one to express that the considered implication holds for any state of any database history whose length is upper bounded by k, is used.

Proving LTL^{fin} Formulae

Prover based on the proof system for "full" first order LTL^{fin} defined by Cerrito, Cialdea-Mayer and Praud for LTL^{fin}. The (tableau-based) calculus in [**?**], has the following properties:

1. When the input contains an initial disequation setting the bound k on the size of temporal frames, the calculus is sound and complete w.r.t. k-validity.

2. When there is no initial disequation, the calculus is sound but (obviously) incomplete w.r.t. validity *tout court*.

We are presently working on the implementation of such a theorem prover.