Logic Programming

Operation Definitions

Michael Genesereth
Computer Science Department
Stanford University
Operations

Queries and View Definitions

\[
\text{goal}(X,Y) :- \ p(X,Y) \ & \ \neg q(Y)
\]

\[
\text{r}(X,Y) :- \ p(X,Y) \ & \ \neg q(Y)
\]

\[
\text{s}(X,Y) :- \ \text{r}(X,Y) \ & \ \text{r}(Y,Z)
\]

Updates and Operation Definitions

\[
p(X) \ & \ \neg q(X) \implies \ \neg p(X) \ & \ q(X)
\]

\[
\text{flip}(X) :: p(X) \ & \ \neg q(X) \implies \ \neg p(X) \ & \ q(X)
\]

\[
\text{flop}(X) :: \text{r}(X,Y) \implies \text{flip}(X) \ & \ \text{flop}(Y)
\]
Syntax
Operation constants represent operations.
  tick - tick of the clock
  click - click a button on a web page
  stack - place one block on another
  mark - place a specific mark in a row and a column

Same spelling conventions as other constants. Like constructors, and predicates, each has a specific arity.

  tick/0
  click/1
  stack/2
  mark/3
An action is an application of an operation to objects.

In what follows, we denote actions using a syntax similar to that of compound terms, viz. an $n$-ary operation constant followed by $n$ terms enclosed in parentheses (as appropriate) and separated by commas.

Examples:
- tick
- click(a)
- stack(a,b)
- mark(x,2,3)

Syntactically, actions are treated as terms.
\[ c(a) :: p(a,b) \land q(a) \implies \neg q(a) \land c(b) \]

- **head**
- **conditions**
- **effects**

\( (action) \)  \( (ordinary \ literals) \)  \( (base \ literals \ or \ actions) \)
Variables

\[ c(X) :: p(X,Y) \land q(X) \implies \neg q(X) \land c(Y) \]
Degenerate Rule

\[ c(X) :: \text{true} \implies \neg p(X) \land q(X) \]

Shorthand

\[ c(X) :: \neg p(X) \land q(X) \]
An operation definition is a finite collection of operation rules with the same operation in the head.

**Example**

\[ c(X) :: p(X) \land q(X) \]
\[ c(X) :: \neg r(X) \implies \neg p(X) \land r(X) \]

A dynamic logic program is a collection of view definitions and operation definitions.
A operation rule is safe if and only if every variable in every literal on the right hand side appears in a positive literal on the left hand side. Also, every variable in a negative literal on the left hand side appears in a prior positive literal.

**Safe Operation Rule**

\[
\text{c}(X) :: \\
\text{p}(X, Y) \& \lnot q(X) \implies \\
\lnot p(X, Y) \& q(X) \& c(Y)
\]

**Unsafe Operation Rule**

\[
\text{c}(X) :: \\
\text{p}(X, Y) \& \lnot q(Z) \implies \\
\lnot p(X, Y) \& q(W) \& c(Y)
\]
Semantics
Given a rule set $\Omega$, the result of applying an action set $\Gamma$ to a dataset $\Delta$ is the dataset that results from *deleting all of the negative effects* of the actions in $\Gamma$ to $\Delta$ and *adding in all of the positive effects*. 
Recall that $E(\Omega, \Delta)$ is the extension of dataset $\Delta$ with respect to ruleset $\Omega$, i.e. the set of true factoids in the vocabulary of the program.

Given a ruleset $\Omega$ with dataset $\Delta$ and a set $\Gamma$ of actions, an instance of an operation rule in $\Omega$ is active if and only if the head of the rule is in $\Gamma$ and the conditions of the rule are all true in $\Delta$ (i.e. all of the positive conditions are in $E(\Omega, \Delta)$ and none of the negative conditions are in $E(\Omega, \Delta)$). Otherwise, the instance is inactive.
Dataset: \{p(a), p(b), p(c), q(a), q(b), q(c), r(b)\}

Ruleset:

\[
\begin{align*}
  s(X) & : = p(X) \land q(X) \\
  u(X) & : : s(X) \land \neg r(X) \implies \neg p(X) \land r(X)
\end{align*}
\]

Actionset: \{u(a), u(b)\}

Extension: \{p(a), p(b), p(c), q(a), q(b), q(c), r(b), s(a), s(b), s(c)\}

Active Instances of operation rule:

\[
\begin{align*}
  u(a) & : : s(a) \land \neg r(a) \implies \neg p(a) \land r(a)
\end{align*}
\]

Inactive Instance:

\[
\begin{align*}
  u(b) & : : s(b) \land \neg r(b) \implies \neg p(b) \land r(b) \\
  u(c) & : : s(a) \land \neg r(c) \implies \neg p(c) \land r(c)
\end{align*}
\]
The expansion of an action set $\Gamma$ with respect to rule set $\Omega$ and dataset $\Delta$ (written $U(\Gamma,\Omega,\Delta)$) is defined as follows.

$\Gamma_0 = \Gamma$

$\Gamma_{n+1} =$ the set of all effects in any active rule instance.

The expansion $U(\Gamma,\Omega,\Delta)$ is the set of all base literals in the fixpoint of this series.

The positive updates (written $A(\Gamma,\Omega,\Delta)$) are the positive base factoids in $U(\Gamma,\Omega,\Delta)$. The negative updates (written $D(\Gamma,\Omega,\Delta)$) are the negative base factoids in $U(\Gamma,\Omega,\Delta)$. 
Dataset: \{p(a), p(b), p(c), q(a), q(b), q(c), r(b)\}

Ruleset:

\[
\text{s}(X) : \neg p(X) \land q(X)
\]

\[
\text{u}(X) :: \text{s}(X) \land \neg r(X) \implies \neg p(X) \land r(X)
\]

Actionset: \{u(a), u(b)\}

Active Instances of operation rules:

\[
\text{u}(a) :: \text{s}(a) \land \neg r(a) \implies \neg p(a) \land r(a)
\]

Expansion: \{\neg p(a), r(a)\}

Negative Updates: \{p(a)\}

Positive Updates: \{r(a)\}
Given rule set $\Omega$, the **result** of applying action set $\Gamma$ to dataset $\Delta$ is the set consisting of all factoids in $\Delta$ minus those in $D(\Gamma,\Omega,\Delta)$ plus those in $A(\Gamma,\Omega,\Delta)$.

$$\Delta - D(\gamma,\Omega,\Delta) \cup A(\gamma,\Omega,\Delta)$$
Dataset: \{p(a), p(b), p(c), q(a), q(b), q(c), r(b)\}

Ruleset:

\[
\begin{align*}
  s(X) & : - p(X) \& q(X) \\
  u(X) & : : s(X) \& \neg r(X) \implies \neg p(X) \& r(X)
\end{align*}
\]

Actionset: \{u(a), u(b)\}

Negative Updates: \{p(a)\}

Positive Updates: \{r(a)\}

Result: \{p(b), p(c), q(a), q(b), q(c), r(a), r(b)\}
A production system is a set of condition-action rules. On each step in the execution of a production system, an active rule is chosen and the actions are performed. The cycle then repeats on the new state.

if \( p(X) \), then del \( p(X) \) and add \( q(X) \)
if \( q(X) \), then del \( q(X) \) and add \( p(X) \)

Before: \( \{ p(a), q(b) \} \)
Step 1: \( \{ q(a), q(b) \} \)
Step 2: \( \{ p(a), q(b) \} \) or \( \{ p(b), q(a) \} \)

When do we stop?
Dynamic logic programs differ from production systems in that all active transition rules “fire” at the same time. (1) All updates are computed before any changes are made, and (2) all changes are made simultaneously.

\[
\text{tick} :: p(X) \implies \neg p(X) \& q(X) \\
\text{tick} :: q(X) \implies \neg q(X) \& p(X)
\]

Before: \{p(a), q(b)\}
After: \{p(b), q(a)\}
Blocks World
External Actions

\[ u(a,b) \]

\[ u(d,e) \]
Unstacking:

\[
\text{tr}(\text{table}(X), \text{do}(u(X,Y), S)) :- \\
\quad \text{tr}(\text{clear}(X), S) \land \text{tr}(\text{on}(X,Y), S)
\]

\[
\text{tr}(\text{clear}(Y), \text{do}(u(X,Y), S)) :- \\
\quad \text{tr}(\text{clear}(X), S) \land \text{tr}(\text{on}(X,Y), S)
\]

Stacking:

\[
\text{tr}(\text{on}(X,Y), \text{do}(s(X,Y), S)) :- \\
\quad \text{tr}(\text{clear}(X), S) \land \text{tr}(\text{table}(X), S) \land \text{tr}(\text{clear}(Y), S)
\]
Frame Axioms for Unstacking

\[
\text{tr(clear(U),do(u(X,Y),S)) } \leftarrow \\
\text{ tr(clear(X),S) } \& \text{ tr(on(X,Y),S) } \& \\
\text{ tr(clear(U),S) }
\]

\[
\text{tr(table(U),do(u(X,Y),S)) } \leftarrow \\
\text{ tr(clear(X),S) } \& \text{ tr(on(X,Y),S) } \& \\
\text{ tr(table(U),S) }
\]

\[
\text{tr(on(U,V),do(u(X,Y),S)) } \leftarrow \\
\text{ tr(clear(X),S) } \& \text{ tr(on(X,Y),S) } \& \\
\text{ tr(on(U,V),S) } \& \text{ distinct(U,X) }
\]

\[
\text{tr(on(U,V),do(u(X,Y),S)) } \leftarrow \\
\text{ tr(clear(X),S) } \& \text{ tr(on(X,Y),S) } \& \\
\text{ tr(on(U,V),S) } \& \text{ distinct(V,Y) }
\]
Frame Axioms for Stacking

\[ \text{tr(clear}(U), \text{do}(s(X,Y), S)) :\]
\[ \text{tr(clear}(X), S) \& \text{tr(table}(X), S) \& \text{tr(clear}(Y), S) \& \text{tr(clear}(U), S) \& \text{distinct}(U,Y) \]

\[ \text{tr(table}(U), \text{do}(s(X,Y), S)) :\]
\[ \text{tr(clear}(X), S) \& \text{tr(table}(X), S) \& \text{tr(clear}(Y), S) \& \text{tr(table}(U), S) \& \text{distinct}(U,X) \]

\[ \text{tr(on}(U,V), \text{do}(s(X,Y), S)) :\]
\[ \text{tr(clear}(X), S) \& \text{tr(table}(X), S) \& \text{tr(clear}(Y), S) \& \text{tr(on}(U,V), S) \]
Describing States

\[
\text{tr(\text{clear(a)},s)} \\
\text{tr(\text{on(a,b)},s)} \\
\text{tr(\text{on(b,c)},s)} \\
\text{tr(\text{on(d,e)},s)} \\
\ldots
\]

\[
\text{do(\text{u(a,b)},s)}
\]

\[
\text{tr(\text{clear(a)}, \text{do(u(a,b)},s))} \\
\text{tr(\text{table(a)}, \text{do(u(a,b)},s))} \\
\text{tr(\text{on(b,c)}, \text{do(u(a,b)},s))} \\
\text{tr(\text{on(d,e)}, \text{do(u(a,b)},s))} \\
\ldots
\]
Describing States

\[ \text{clear}(a) \]
\[ \text{on}(a,b) \]
\[ \text{on}(b,c) \]
\[ \text{on}(d,e) \]
\[ \ldots \]

\[ u(a,b) \]

\[ \text{clear}(a) \]
\[ \text{table}(a) \]
\[ \text{clear}(b) \]
\[ \text{on}(b,c) \]
\[ \text{on}(d,e) \]
\[ \ldots \]
Operations:

\( u(x, y) \) means that \( x \) is moved from \( y \) to the table.
\( s(x, y) \) means that \( x \) is moved from the table to \( y \).

Operation Definitions:

\[ u(X,Y) :: \]
\[ \text{clear}(X) \land \text{on}(X,Y) \]
\[ \Rightarrow \neg \text{on}(X,Y) \land \text{table}(X) \land \text{clear}(Y) \]

\[ s(X,Y) :: \]
\[ \text{table}(X) \land \text{clear}(X) \land \text{clear}(Y) \]
\[ \Rightarrow \neg \text{table}(X) \land \neg \text{clear}(Y) \land \text{on}(X,Y) \]
Graphs
Example

Dataset:

- edge(a,b)
- edge(b,d)
- edge(b,e)

Operations:

- copy(X,Y) - copy outgoing links from X to Y.
- invert(X) - reverse the incoming arcs to X.
- insert(X,Y) - add arc from X to Y and all Y successors.
Dataset:
{edge(a, b), edge(b, d), edge(b, e)}

Operation Definition:
copy(X, Y) :: edge(X, Z) ==> edge(Y, Z)

Action: copy(b, c)

Positive Updates: {edge(c, d), edge(c, e)}
Negative Updates: {}

Result:
{edge(a, b), edge(b, d), edge(b, e), edge(c, d), edge(c, e)}
Dataset:
{edge(a,b), edge(b,d), edge(b,e), edge(c,d), edge(c,e)}

Operation Definition:
invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

Action: invert(c)

Positive Updates: {edge(d,c), edge(e,c)}
Negative Updates: {edge(c,d), edge(c,e)}

Result:
{edge(a,b), edge(b,d), edge(b,e), edge(d,c), edge(e,c)}
Dataset:

\{\text{edge}(a,b), \text{edge}(b,d), \text{edge}(b,e), \text{edge}(d,c), \text{edge}(e,c)\}\n
Operation Definition:

\begin{align*}
\text{insert}(X,Y) &:: \text{edge}(X,Y) \\
\text{insert}(X,Y) &:: \text{edge}(Y,Z) \implies \text{insert}(X,Z)
\end{align*}

Action: \text{insert}(w,b)

Expansion:

\{\text{insert}(w,b), \text{edge}(w,b), \text{insert}(w,d), \text{insert}(w,e), \\
\text{edge}(w,d), \text{edge}(w,e), \text{insert}(w,c), \text{edge}(w,c)\}\n
Negative Updates: {}

Positive Updates:

\{\text{edge}(w,b), \text{edge}(w,d), \text{edge}(w,e), \text{edge}(w,c)\}\n
Example
Lambda

edge(a, b)
edge(b, d)
edge(b, e)

Library

copy(X, Y) :: edge(X, Z) => edge(Y, Z)

invert(X) :: edge(X, Y) => ~edge(X, Y) & edge(Y, X)

insert(X, Y) :: edge(X, Y)
insert(X, Y) :: edge(Y, Z) => insert(X, Z)
Lambda

edge(a,b)
edge(b,d)
edge(b,e)

Library

\[
\begin{align*}
copy(X,Y) & : \ edge(X,Z) \implies edge(Y,Z) \\
invert(X) & : \ edge(X,Y) \implies \neg edge(X,Y) \land edge(Y,X) \\
insert(X,Y) & : \ edge(X,Y) \\
insert(X,Y) & : \ edge(Y,Z) \implies insert(X,Z)
\end{align*}
\]
Lambda

edge(a,b)
edge(b,d)
edge(b,e)
edge(c,d)
edge(c,e)

Library

copy(X,Y) :: edge(X,Z) => edge(Y,Z)
invert(X) :: edge(X,Y) => ~edge(X,Y) & edge(Y,X)
insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) => insert(X,Z)
Lambda
- edge(a,b)
- edge(b,d)
- edge(b,e)
- edge(c,d)
- edge(c,e)

Execute
- Action: invert(c)
- Expand on update: Off
- Run on clock tick: Off
- ~edge(c,d)
- edge(d,c)
- ~edge(c,e)
- edge(e,c)

Library
- copy(X,Y) :: edge(X,Z) => edge(Y,Z)
- invert(X) :: edge(X,Y) => ~edge(X,Y) & edge(Y,X)
- insert(X,Y) :: edge(X,Y)
- insert(X,Y) :: edge(Y,Z) => insert(X,Z)
Lambda

edge(a,b)
edge(b,d)
edge(b,e)
edge(d,c)
edge(e,c)

Execute
Action: invert(c)

Library
Save
Revert

copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
Lambda

edge(a,b)
edge(b,d)
edge(b,e)
edge(d,c)
edge(e,c)

Execute

Action insert(w,b)

edge(w,b)
edge(w,d)
edge(w,c)
edge(w,e)

Expand
- Expand on update
- Run on clock tick

Library

copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ¬edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
edge(a,b)
edge(b,d)
edge(b,e)
edge(d,c)
edge(e,c)
edge(w,b)
edge(w,d)
edge(w,c)
edge(w,e)

copy(X,Y) :: edge(X,Z) => edge(Y,Z)

invert(X) :: edge(X,Y) => ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) => insert(X,Z)
Constraint Enforcement
Requests:
  \texttt{add(p(a))} is a request to add \texttt{p(a)} to the database.
  \texttt{del(p(a))} is a request to remove \texttt{p(a)} from the database.

Simultaneous updates okay (*Sierra supports singles only*)
  \texttt{del(p(a))}
  \texttt{del(q(b))}
  \texttt{add(p(b))}
  \texttt{add(q(a))}

NB: DB system may or may not act upon requests. Might reject, might accept as is, might modify.
Suppose we have a database and a set of constraints and suppose the user requests an update which, if applied, would produce a dataset that violates the constraints.

Dataset:  \{\text{dead(art)}, \text{dead(chris)}, \text{alive(bea)}\}
Constraint: \text{illegal :- alive(X) \& dead(X)}
Request: \text{add(alive(chris))}

Alternatives:
- Reject the update
- **Repair the update**
- Something else???
A *mutual exclusion constraint* states that every object falls into exactly one of a collection of disjoint classes.

**Dataset:** \{dead(art), dead(chris), alive(bea)\}

**Constraint:** illegal :- alive(X) & dead(X)

**Ruleset:**

- add(alive(X)) :: ~dead(X)
- add(dead(X)) :: ~alive(X)

**Request:** add(alive(chris))

**Result:** \{dead(art), alive(bea), alive(chris)\}
A functional dependency states that a relationship has one and only one value for each argument, e.g. year in school.

**Dataset:** \{year(a,1), year(b,2), year(c,3)\}

**Constraint:**

illegal :- year(X,M)&year(X,N)&distinct(M,N)

**Ruleset:**

add(year(S,Y)) :: year(S,O) ==> ~year(S,O)
add(year(S,Y)) :: year(S,Y)

**Request:** add(year(b,3))

**Result:** year(a,1), year(b,3), year(c,3)\}

**Alternative:**

add(year(S,Y)) ::

year(S,O) ==> ~year(S,O) & year(S,Y)
An inclusion dependency states that, if an object appears as an argument of one predicate, then it must also appear in a specific position in another predicate.

**Dataset:** \{adult(art), p(art, bea)\}

**Constraint:** illegal :- p(X, Y) & ~adult(X)

**Ruleset:** add(p(X, Y)) :: adult(X)

**Request:** add(p(bea, coe))

**Result:**
\{adult(art), adult(bea), p(art, bob), p(bea, coe)\}
Materialization
A **materialized view** is a view relation that is stored explicitly in the database.

**Ruleset:**

\[
\text{grandparent}(X,Z) \leftarrow \text{parent}(X,Y) \land \text{parent}(Y,Z)
\]

**Dataset:**

- parent(art,bob)
- parent(art,bea)
- parent(bob,cal)
- parent(bob,cam)
- parent(bea,cat)
- parent(bea,coe)
- grandparent(art,cal)
- grandparent(art,cam)
- grandparent(art,cat)
- grandparent(art,coe)
Ruleset

\[ s(X,Y,Z) :- r(X) \land r(Y) \land r(Z) \]
\[ r(X) :- p(X,Y) \land p(Y,Z) \]

Computation Cost for \( s \)

- \( r \) computed multiple times
- For \( n=2 \), unifications = 1242
- For \( n=3 \), unifications = 41636
- Where \( n \) is the number of objects

Storage for \( p \)

- \( O(n^2) \) facts stored
Example with $s$ Materialized

Ruleset

\[
\begin{align*}
s(X,Y,Z) & : \neg r(X) \land r(Y) \land r(Z) \\
r(X) & : \neg p(X,Y) \land p(Y,Z)
\end{align*}
\]

Computation Cost for $s$

- for $n=2$, unifications = 8
- for $n=3$, unifications = 27

Storage for $s$

- $O(n^3)$ in worst case
Ruleset

\[ s(X,Y,Z) :- r(X) \& r(Y) \& r(Z) \]
\[ r(X) :- p(X,Y) \& p(Y,Z) \]

Computation Cost for \( s \)

for \( n=2 \), unifications = 15
for \( n=3 \), unifications = 40
where \( n \) is the number of objects

Computation Cost for \( r \)

for \( n=2 \), unifications = 17
for \( n=3 \), unifications = 55

Storage for \( r \)

\( O(n) \) in worst case
Conversion process. (1) Rename in all rules defining view to be materialized and replace in all other rules. (2) Write action definitions to relate old and new name.

**Old View Definitions**
\[
\begin{align*}
  s(X,Y,Z) & : = r(X) \& r(Y) \& r(Z) \\
r(X) & : = p(X,Y) \& p(Y,Z)
\end{align*}
\]

**New View Definitions**
\[
\begin{align*}
  s(X,Y,Z) & : = rr(X) \& rr(Y) \& rr(Z) \\
r(X) & : = p(X,Y) \& p(Y,Z)
\end{align*}
\]

**Operation Definitions**
\[
\begin{align*}
  \text{add}(p(U,V)) & :: rr(X) \implies \neg rr(X) \\
  \text{del}(p(U,V)) & :: rr(X) \implies \neg rr(X) \\
  \text{add}(p(U,V)) & :: r(X) \implies rr(X) \\
  \text{del}(p(U,V)) & :: r(X) \implies rr(X)
\end{align*}
\]
Old Base Data: \{p(a,b), p(b,a), p(c,d)\}
Old Materialized Data: \{rr(a), rr(b)\}

Ruleset:
\[
\begin{align*}
  r(X) & : \text{=} \ p(X,Y) \ & \text{&} \ p(Y,Z) \\
  \text{add}(p(U,V)) & : \ rr(X) \ \Longrightarrow \ \neg rr(X) \\
  \text{del}(p(U,V)) & : \ rr(X) \ \Longrightarrow \ \neg rr(X) \\
  \text{add}(p(U,V)) & : \ r(X) \ \Longrightarrow \ rr(X) \\
  \text{del}(p(U,V)) & : \ r(X) \ \Longrightarrow \ rr(X)
\end{align*}
\]

Changes: \{\text{del}(p(a,b)), \text{add}(p(d,c))\}

Negative Update: \{rr(a), rr(b)\}

Positive Update: \{rr(c), rr(d)\}

New Base Data: \{p(b,a), p(c,d), p(d,c)\}
New Materialized Data: \{rr(c), rr(d)\}
Conversion process. (1) Rename in all rules defining view to be materialized and replace in all other rules. (2) Write transition rules to relate old and new name. (3) Remove rules defining the materialized view and replace with differential rules relating changes to base relations and changes to views.
Old View Definitions
\[
\begin{align*}
s(X,Y,Z) & :\equiv r(X) \& r(Y) \& r(Z) \\
r(X) & :\equiv p(X,Y) \& p(Y,Z)
\end{align*}
\]

New View Definitions
\[
\begin{align*}
s(X,Y,Z) & :\equiv rr(X) \& rr(Y) \& rr(Z)
\end{align*}
\]

Operation Definitions
\[
\begin{align*}
\text{add}(p(X,Y)) & :\equiv p(Y,Z) \Rightarrow rr(X) \\
\text{add}(p(Y,Z)) & :\equiv p(X,Y) \Rightarrow rr(X)
\end{align*}
\]

Exercise for reader: What are the deletion rules?
Old Dataset: \{p(a,b), p(b,a), rr(a), rr(b)\}

Ruleset:

- \text{add}(p(X,Y)) \land p(Y,Z) \land \neg \text{del}(p(Y,Z)) \implies rr(X)
- p(X,Y) \land \neg \text{del}(p(X,Y)) \land \text{add}(p(Y,Z)) \implies rr(X)
- \text{add}(p(X,Y)) \land \text{add}(p(Y,Z)) \implies rr(X)

Change Request: \{\text{add}(p(c,d)), \text{add}(p(d,c))\}

Negative Update: \{

Positive Update: \{rr(c), rr(d)\}

New Dataset:

\{p(a,b), p(b,a), p(c,d), p(d,c),
     rr(a), rr(b), rr(c), rr(d)\}
Differential Relational Logic

Differential Relational Calculus for Integrity Maintenance

Levent V. Orman

Abstract — A differential calculus for first-order logic is developed to enforce database integrity. Formal differentiation of first-order sentences is useful in maintaining database integrity, since once a database constraint is expressed as a first-order sentence, its derivative with respect to a transaction provides the necessary and sufficient condition for maintaining integrity. The derivative is often much simpler to test than the original constraint since it maintains integrity differentially by assuming integrity before the transaction, and testing only for new violations. The formal differentiation requires no resolution search, but only substitution. It is more efficient than resolution-based approaches; and it provides a considerably more general solution than previous substitution-based methods since it is valid for all first-order sentences and with all transactions involving arbitrary collections of atomic changes to the database. It also produces a large number of sufficient conditions that are often less strict than those of the previous approaches; and it can be extended to accommodate many dynamic constraints.

Index Terms — Database integrity, integrity maintenance, relational database, first-order logic, formal differentiation.

1 INTRODUCTION

DATABASE integrity constraints are logical statements that define valid states of the database. They must be obeyed by the data at all times to ensure the validity of data. They are critical in maintaining the quality of data and preventing a variety of errors and abuses. Efficient maintenance of integrity is a critical problem since testing the validity of a large number of constraints against a large database, and after each transaction that may change the database state is an expensive undertaking. Considerable efficiency can be gained by maintaining integrity differentially, by assuming integrity before the transaction, and testing for new violations only [10], [11]. The critical problem in differential integrity maintenance is deriving the test conditions for these new violations. These conditions are often considerably easier to test than the original constraint, but they are also considerably different from the original constraint, and their derivation is not trivial, adding to the cost of integrity maintenance. Moreover, these test conditions are different for every transaction and, although they could be derived and compiled ahead of time for anticipated transactions, ad hoc transactions have even more stringent efficiency requirements—since their test conditions have to be derived in real time after the transaction is received and before it can be authorized [8], [13].

There have been two general approaches to the derivation of differential integrity test conditions. The first approach involves studying the quantifier structure of the constraint, and substituting the update values of the transactions for some constraint variables [1], [6], [10], [15]. This approach is very efficient in deriving the test conditions, but it is severely restrictive. The general problem is the difficulty of finding a procedure for analyzing the structure of arbitrarily complex constraints and arbitrarily complex transactions. Consequently, each implementation has its own severe restrictions. Some fail to derive the test conditions for constraints with complex quantifier structures or for constraints with multiple occurrences of a relation. Some find only one test condition when there are alternative conditions that may be easier to test. For complex transactions with multiple updates, this approach either fails or requires repeated application of the substitution algorithm. It also often derives conditions that need to be tested after the execution of the transaction, requiring an expensive rollback if the integrity test fails. The second approach involves resolution theorem proving where, given the constraint is satisfied before the transaction and the database state is changed by a transaction, one needs to prove that the constraint continues to be satisfied [8], [16], [19]. This approach is very general. It derives a necessary and sufficient test condition for all first-order sentences that satisfy the domain independence requirement, and it can be used with arbitrary nonatomic transactions. This approach also finds some sufficient conditions that may be easier to test; and its tests can be applied before the transaction, eliminating the need for a rollback. The disadvantages of this approach are the need for resolution search that makes it less efficient than substitution, and the need to write complex transition axioms to describe transactions. Both of these problems are partially alleviated for anticipated transactions by performing these tasks ahead of time and compiling the results into update programs; but for ad hoc transactions, real time integrity maintenance remains a problem. A similar approach was described in [15] where a differential relational algebra was developed to compute the changes to a relational algebra expression. This approach is also very general since it
DRC as Dynamic Logic Program

**Ruleset**

\[ s(X) :- p(X) \land q(X) \]

**Differentials** (for single actions):

\[ s^+(X) :- p^+(X) \land q(X) \]
\[ s^+(X) :- p(X) \land q^+(X) \]
\[ s^-(X) :- p^-(X) \]
\[ s^-(X) :- q^-(X) \]

**Operation Definitions:**

\[ \text{add}(p(X)) :: \land q(X) \implies s(X) \]
\[ \text{add}(q(X)) :: p(X) \implies s(X) \]
\[ \text{del}(p(X)) :: \neg s(X) \]
\[ \text{del}(q(X)) :: \neg s(X) \]
Update Through Views
View Definition

\[ r(X) :- p(X) \]
\[ r(X) :- q(X) \]

Request: add( r(bob) )

Positive Update:

\{p(bob)\}?
\{q(bob)\}?
\{p(bob), q(bob)\}?
\{\}\?

*What if p is dead and q is alive?*
Update Policies

View Definition
\[
\begin{align*}
  r(X) & : = p(X) \\
  r(X) & : = q(X)
\end{align*}
\]

Update Rule
\[
\text{add}(r(X)) : : \neg q(X) \implies p(X)
\]