Logic Programming
Dynamic Logic Programs

Michael Genesereth
Computer Science Department
Stanford University
Transform allows users to specify one-shot transformations.

Transition Rules allows users to instruct the system to perform transformations automatically.
Transition Rules
\[ \neg p(X, Y) \land \neg q(Y) \implies \neg p(X, Y) \land q(Y) \]

\textit{conditions} \quad \textit{effects}
A transition 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 Transition Rule**

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

**Unsafe Transition Rules**

\[ p(X) \land \neg q(X) \implies \neg p(X) \land q(Y) \]
\[ p(X) \land \neg q(Y) \implies \neg p(X) \land q(X) \]
A dynamic logic program (DLP) is a finite collection of view definitions and transition rules.

Example

\[ s(X) \leftarrow p(X) \land q(X) \]
\[ s(X) \land \neg r(X) \implies \neg p(X) \land r(X) \]

NB: In purposes of analysis, we sometimes talk about infinite DLPs, e.g. the set of all instances of rules.
Recall that $E(\Omega, \Delta)$ is the extension of dataset $\Delta$ with respect to ruleset $\Omega$.

Given a ruleset $\Omega$ with dataset $\Delta$, an instance of a transition rule $t$ in $\Omega$ is active if $E(\Omega, \Delta)$ contains all of the rule’s positive conditions and none of its negative conditions. Otherwise, the instance is inactive.
Ruleset

\[ s(X) :- p(X) \& q(X) \]
\[ s(X) \& \neg r(X) \implies \neg p(X) \& r(X) \]

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

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

Active Instances

\[ s(a) \& \neg r(a) \implies \neg p(a) \& r(a) \]
\[ s(c) \& \neg r(c) \implies \neg p(c) \& r(c) \]

Inactive Instance

\[ s(b) \& \neg r(b) \implies \neg p(b) \& r(b) \]
The **positive update** $add(t, \Delta)$ of applying a *transition rule* $t$ to $\Delta$ is the set of all positive effects in any active instance of $t$, and the **negative update** $del(t, \Delta)$ is the set of all negative effects in any active instance of $t$.

The **positive update** $add(\Omega, \Delta)$ of applying a *dynamic logic program* $\Omega$ to $\Delta$ is the set of all positive updates for all transition rules in $\Omega$, and the **negative update** $del(\Omega, \Delta)$ is the set of all negative updates for all transition rules in $\Omega$. 
Ruleset

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

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

Active Instances

\[
\begin{align*}
  s(a) \ & \ \sim r(a) & \implies \ \sim p(a) \ & \ r(a) \\
  s(c) \ & \ \sim r(c) & \implies \ \sim p(c) \ & \ r(c)
\end{align*}
\]

Positive Update: \{r(a), r(c)\}

Negative Update: \{p(a), p(c)\}
The **update** \( u(\Omega, \Delta) \) of \( \Delta \) with respect to \( \Omega \) is the set consisting of all factoids in \( \Delta \) minus those in \( del(\Omega, \Delta) \) plus those in \( add(\Omega, \Delta) \).

\[
u(\Omega, \Delta) = \Delta - del(\Omega, \Delta) \cup add(\Omega, \Delta)
\]

**Dataset:** \( \{p(a), p(b), p(c), q(a), q(b), q(c), r(b)\} \)

**Positive Update:** \( \{r(a), r(c)\} \)

**Negative Update:** \( \{p(a), p(c)\} \)

**Result:** \( \{p(b), q(a), q(b), q(c), r(a), r(b), r(c)\} \)
An execution of a ruleset $\Omega$ on a dataset $\Delta$ is a dataset sequence $\Delta_1, \Delta_2, \ldots$, where $\Delta_1 = \Delta$ and $\Delta_{n+1} = u(\Omega, \Delta_n)$. 
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), p(b)\}
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.

\[
\begin{align*}
\text{if } p(X), \text{ then del } p(X) \text{ and add } q(X) \\
\text{if } q(X), \text{ then del } q(X) \text{ and add } p(X)
\end{align*}
\]

Before: \{p(a), q(b)\}
After: \{p(b), q(a)\}
Example - The Game of Life
(1) Any live cell with *two or three* live neighbors lives on to the next generation.

(2) Any live cell with *fewer than two* live neighbors dies (as if caused by underpopulation).

(3) Any live cell with *more than three* live neighbors dies (as if by overpopulation).

(4) Any dead cell with *exactly three* live neighbors becomes a live cell (as if by reproduction).
Symbols: \(c_{11}, c_{12}, \ldots\)

Unary Predicates:
- \(\text{on} - \) cell is live
- \(\text{cell} - \) true of cells

Binary Predicates:
- \(\text{neighbor} - \) cells are neighbors
- \(\text{supports} - \) first cell is alive and a neighbor of the second

Definition of support
\[
\text{supports}(X,Y) \; :\; \text{neighbor}(X,Y) \; \& \; \text{on}(X)
\]
Transition Rules

Any live cell with fewer than two live neighbors dies.

\[
\text{on}(Y) \land \text{countofall}(X, \text{supports}(X,Y), N) \land \text{leq}(N, 1) \implies \neg \text{on}(Y)
\]

Any live cell with more than three live neighbors dies.

\[
\text{on}(Y) \land \text{countofall}(X, \text{supports}(X,Y), N) \land \text{geq}(N, 4) \implies \neg \text{on}(Y)
\]

Any dead cell with exactly three live neighbors becomes a live cell.

\[
\text{cell}(Y) \land \neg \text{on}(Y) \land \text{countofall}(X, \text{supports}(X,Y), 3) \implies \text{on}(Y)
\]
http://logicprogramming.stanford.edu/examples/gameoflife.html
Example - Blocks World
Blocks World
External Actions

\[ u(a,b) \quad u(d,e) \]
Symbols: a, b, c, d, e

Unary Predicates:
  clear - blocks with no blocks on top
  table - blocks on the table

Binary Predicates:
  on - pairs of blocks in which first is on the second
  above - pairs in which first block is above the second

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.
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 \( y \).

Transition Rules:

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

\[ s(X,Y) \land \text{table}(X) \land \text{clear}(X) \land \text{clear}(Y) \implies \neg\text{table}(X) \land \neg\text{clear}(Y) \land \text{on}(X,Y) \]
Dataset:

- `clear(c)`
- `on(c,a)`
- `table(b)`
- `clear(d)`
- `on(a,b)`
- `table(e)`
- `on(d,e)`

Transition Rule:

\[
u(X,Y) \& clear(X) \& on(X,Y) \\
===> \lnot on(X,Y) \& table(X) \& clear(Y)\]

Result of `u(c,a)`:

- `clear(a)`
- `on(a,b)`
- `table(b)`
- `clear(c)`
- `on(d,e)`
- `table(c)`
- `clear(d)`
- `table(e)`
Example - Database Systems
Requests:
- \(\text{add}(p)\) is a request to add \(p\) to the database.
- \(\text{del}(p)\) is a request to remove \(p\) from the database.

Simultaneous updates okay
- \(\text{del}(p(a))\)
- \(\text{del}(q(b))\)
- \(\text{add}(p(b))\)
- \(\text{add}(q(a))\)

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, when applied, produces a dataset that violates the constraints.

Action 1: Reject the update

Action 2: Repair the update
A partition constraint states that every object falls into exactly one of a collection of disjoint classes.

**Ruleset**

- `add(female(X)) ==> ~male(X)`
- `add(male(X)) ==> ~female(X)`

**Dataset:** `{male(art), male(chris), female(bea)}`

**Request:** `add(female(chris))`

**Result:** `{male(art), female(bea), female(chris)}`
A functional dependency states that a relationship has one and only one value for each argument, e.g. father, mother, year in school, etc.

**Ruleset**

- $\text{add(year(S,Y))} \& \text{year(S,O)} \Rightarrow \neg\text{year(S,O)}$
- $\text{add(year(S,Y))} \Rightarrow \text{year(S,Y)}$

- $\text{add(year(S,Y))} \& \text{year(S,O)}$
  $\Rightarrow \neg\text{year(S,O)} \& \text{year(S,Y)}$

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

**Request:** add\(\text{year(b,3)}\)

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

**Ruleset**

\[
\text{add}(p(X,Y)) \implies \text{adult}(X)
\]

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

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

**Result:**

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

**Ruleset**

\[
\text{father}(X,Y) \leftarrow \text{parent}(X,Y) \land \text{male}(X)
\]
add(parent(X,Y)) & male(X) & ~del(male(X))
   ==> father(X,Y)

parent(X,Y) & add(male(X)) & ~del(parent(X,Y))
   ==> father(X,Y)

add(parent(X,Y)) & add(male(X))
   ==> father(X,Y)

del(parent(X,Y)) ==> ~father(X,Y)

del(male(X)) ==> ~father(X,Y)
Update Through Views

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

Request: \texttt{add}(r(bob))

Positive Update:
\[
  \{p(bob)\}?
  \{q(bob)\}?
  \{p(bob), q(bob)\}? \quad \text{What if } p \text{ is male and } q \text{ is female?}
  \{\}\?
\]
View Definition

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

Update Rule

\[ \text{add}(r(X)) \& \neg q(X) \implies p(X) \]