Dynamic Logic Programming
Extended Abstract

Michael Genesereth
Computer Science Department
Stanford University

1. Introduction

Dynamic Logic Programming (DLP) is an extension to logic programming designed to support the representation of knowledge about dynamic worlds. It combines the strengths of safe, stratified, side-effect-free logic programming in defining relations with the power of simultaneous transition rules for defining dynamic operations. Because relation definitions in DLP are safe and stratified and side-effect-free, dynamic logic programs are simpler than general Prolog programs and they allow for efficient implementation. At the same time, defining operations using simultaneous transition rules adds expressive power without compromising the conceptual simplicity of logic programming. DLP is the basis for the logic programming language Epilog (aka Dynamic Prolog).

In Dynamic Logic Programming, the states of the application environment are modeled as sets of ground atomic propositions (here called datasets), and additional information is expressed in the form of rules that can be applied to these instances. View definitions define higher level view relations in terms of lower level base relations, and operation definitions specify how the world state changes in response to external inputs (such as the actions of agents or the passage of time).

Views are defined by writing Prolog-style rules. For example, the rule below says that $g$ is true of $x$ and $z$ if there is a $y$ such that $p$ is true of $x$ and $y$ and $p$ is also true of $y$ and $z$.

$$g(X,Y) :- p(X,Y) & p(Y,Z)$$

Operations are defined using transition rules. For example, the following rule says that when the
operation \( a \) is applied to \( x \) in a state, the for any \( y \) such that \( p \) holds of \( x \) and \( y \) then \( p \) will be false of \( x \) and \( y \) in the next state and \( p \) will be true of \( y \) and \( x \).

\[
a(X) :: p(X,Y) ==> \neg p(X,Y) \& p(Y,X)
\]

This document is a brief overview of dynamic logic programming. It introduces the key elements in the form of an example. The full paper associated with this overview gives formal details and a comparison of dynamic logic programming with other approaches to formalizing dynamics.

2. Example - Tic Tac Toe

As an example of a dynamic logic programming, consider the task of formalizing the rules for the game of Tic Tac Toe (also called Noughts and Crosses, Xs and Os). In what follows, we show how to represent game states as datasets; we show how to define properties of states using view definitions; and we show how to define "moves" in the game using operation definitions.

Tic Tac Toe is a game for two players (the X player and the O player) who take turns placing their marks in a 3x3 grid. The first player to place three of his marks in a horizontal, vertical, or diagonal row wins the game.

In our definition of Tic Tac Toe, states are characterized by the cell contents and control (whose turn it is to play). In what follows, we use the ternary relation constant cell together with a row \( m \) and a column \( n \) and a mark \( w \) to designate the fact that the cell in row \( m \) and column \( n \) contains \( w \) where \( w \) is either an \( x \) or an \( o \) or a \( b \) (for blank). We use the unary relation constant control to state that it is that role's turn to mark a cell. Using this vocabulary, we can describe the game state shown on the left below using the dataset shown on the right.

```
X O                  X O
cell(1,1,x)          cell(1,2,o)
cell(1,3,b)          cell(2,1,b)
cell(2,2,x)          cell(2,3,o)
cell(3,1,b)          cell(3,2,b)
cell(3,3,b)          control(x)
```

The first step in formalizing the rules of the game is to define legality of moves. A player may mark a cell if that cell is blank. Otherwise, it has no legal actions.

\[
\text{legal}(M,N) :- \text{cell}(M,N,b)
\]

Next, we define how the state changes in response to the performance of legal actions. If a player that has control and marks a cell, the cell is then marked and control switches to the other player.

```
mark(M,N) :: control(Z) ==> \neg \text{cell}(M,N,b) \& \text{cell}(M,N,Z)
mark(M,N) :: control(x) ==> \neg \text{control}(x) \& \text{control}(o)
mark(M,N) :: control(o) ==> \neg \text{control}(o) \& \text{control}(x)
```

Finally, to complete our game description, we define some properties of game states - rows,
columns, diagonals, lines - and we must say when the game terminates.

A row of marks mean that there are three marks all with the same first coordinate. The column and diagonal relations are defined analogously.

\[
\begin{align*}
\text{row}(M,Z) & :\text{- cell}(M,1,Z) & & \& & \text{cell}(M,2,Z) & & \& & \text{cell}(M,3,Z) \\
\text{column}(M,Z) & :\text{- cell}(1,N,Z) & & \& & \text{cell}(2,N,Z) & & \& & \text{cell}(3,N,Z) \\
\text{diagonal}(Z) & :\text{- cell}(1,1,Z) & & \& & \text{cell}(2,2,Z) & & \& & \text{cell}(3,3,Z) \\
\text{diagonal}(Z) & :\text{- cell}(1,3,Z) & & \& & \text{cell}(2,2,Z) & & \& & \text{cell}(3,1,Z)
\end{align*}
\]

A line is a row of marks of the same type or a column or a diagonal.

\[
\begin{align*}
\text{line}(Z) & :\text{- row}(M,Z) \\
\text{line}(Z) & :\text{- column}(M,Z) \\
\text{line}(Z) & :\text{- diagonal}(Z)
\end{align*}
\]

A game is over whenever either player has a line of marks of the appropriate type or if there are no blank cells. We define the 0-ary relation \text{open} here to mean that there is at least one blank cell.

\[
\begin{align*}
\text{terminal} & :\text{- line}(x) \\
\text{terminal} & :\text{- line}(o) \\
\text{terminal} & :\text{- open} \\
\text{open} & :\text{- cell}(M,N,b)
\end{align*}
\]

Note that the rules given here describe the states of the game and specify the game's "physics" (in this case, how it reacts to external inputs). The rules do not specify how to play the game effectively. In order to decide this, a player needs to consider the effects of its legal moves in order to decide a course of action that will lead to a line of his marks while considering the possible moves of the other player.

### 3. Comparison to Other Approaches

Over the years, various LP researchers, have developed extensions to deal with dynamics, e.g. assert and retract in standard Prolog, production systems, active databases, transactions in Transaction Logic, constraint handling rules in CHR, evolving logic programs in EVOLP, and reactive rules in DALI and LPS. In the full paper, we summarize these approaches and highlight their commonalities and differences. We mention just two other approaches here.

Prolog's assert and retract provide one way to model dynamics. The key is a conceptualization of dynamics as destructive change of state - states are modeled as sets of stored facts, and changes to state are modeled as applications of assert and retract to these sets of facts. Unfortunately, the semantics of logic programs involving assert and retract is unsatisfying because of the way the execution of these actions gets mixed up with query evaluation in the standard Prolog interpreter. Dynamic logic programming cleans things up by separating the formalization of dynamics from the definition of relations using standard Prolog rules.

Production systems are another way of expressing dynamics. The transition rules used to define operations in DLP are similar, but there are some important differences. In most production systems, only one rule is applied at a time. (Many rules may be "triggered", but typically only one is "fired".) In dynamic logic programs, all transition rules are executed simultaneously, and all updates (both deletions and additions) are applied to the dataset before the rules fire again. This simplifies the specification of dynamics in many cases, and avoids many problems endemic to sequential update systems, such as unintended race conditions and deadlocks.
References


