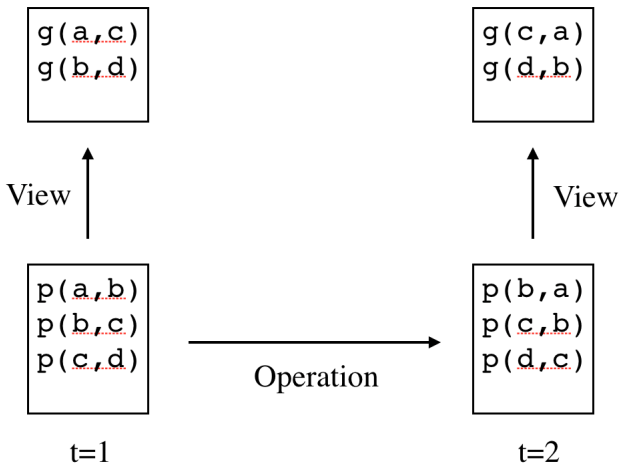
Dynamic Logic Programming

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) ==> ~p(X,Y) & p(Y,X)
```

This paper provides technical detail about the syntax and semantics of Dynamic Logic Programming. Section 2 describes the concept of datasets; Section 3 gives details of view rules; and Section 4 covers transition rules and shows how they are used in formalizing dynamic behavior. Section 5 gives an example of the use of Dynamic Logic Programming in defining the game of Tic Tac Toe.

2. Datasets

A *vocabulary* is a collection of *object constants*, *function constants*, and *relation constants*. Each function constant and relation constant has an associated *arity*, i.e. the number of *arguments* allowed in any expression involving that constant.

A *ground functional term* is an expression formed from an *n*-ary function constant and *n* ground terms. In this paper, we write functional terms in traditional mathematical notation - the function constant followed by its *arguments* enclosed in parentheses and separated by commas. For example, if `f` is a unary function constant and `a` is an object constant, then `f(a)`, `f(f(a))`, and `f(f(f(a)))` are all ground functional terms. A *ground term* is either an object constant or a ground functional term.

A *ground atom* (or *factoid*) is an expression formed from an *n*-ary relation constant and *n* ground terms. In analogy with functional terms, we write factoids in traditional mathematical notation - the relation constant followed by its *arguments* enclosed in parentheses and separated by commas. For example, if `r` is a binary relation constant and `a` and `b` are object constants, then `r(a,b)` is a factoid.

The *Herbrand universe* for a given vocabulary is the set of all ground terms that can be formed from the constants in the vocabulary. In the absence of function constants, the Herbrand universe for a vocabulary is just the set of all object constants. In the presence of function constants with arity greater than 0, the Herbrand universe is necessarily infinite, as it includes not just object constants but also functional terms nested arbitrarily deeply.

The *Herbrand base* for a database is the set of all factoids that can be formed from the constants in its vocabulary. For example, for a vocabulary with just two object constants `a` and `b` and a single binary relation constant `r`, the Herbrand base is {`r(a,a)`, `r(a,b)`, `r(b,a)`, `r(b,b)`}.

A *dataset* is any subset of the Herbrand base, i.e. an arbitrary set of the factoids that can be formed from the vocabulary of the database. The factoids in a dataset representing a state are typically assumed to be true in that state, and all other factoids in the Herbrand base are typically assumed to be false.

3. View Definitions

A *static logic program* is a set of rules that define new relations in terms of existing relations. Such view definitions take the form of Prolog-like rules with the constraint that the rules are safe and stratified and side-effect-free.

The vocabulary of a static logic program is a superset of the vocabulary of any dataset to which it is applied. It includes the object, function, and relation constants used in the dataset, but it can include additional object, function, and relation constants as well.

Static logic programs can also include a new type of symbol, called a *variable*. Variables allow us to state relationships among objects without naming specific objects. In what follows, we write variables as strings of alphanumeric characters beginning with a capital letter, e.g. `X`, `Y`, `Z`, `Mass`, `Speed`, and so forth.

Atoms are analogous to dataset factoids except that they can optionally contain variables as well as object constants. For example, if `r` is a binary relation constant, if `a` is an object constant, and if `X` and `Y` are variables, then `r(a,X)` is an atom, as is `r(a,Y)` and `r(X,Y)` and `r(X,X)`.

A *literal* is either an atom or a negation of an atom (i.e. an expression stating that the atom is false). A simple atom is called a *positive* literal, The negation of an atom is called a *negative* literal. In what follows, we write negative literals using the negation sign `~`. For example, if `p(a,b)` is an atom, then `~p(a,b)` denotes the negation of this atom.

A *rule* is an expression consisting of a distinguished atom, called the head, and zero or more atoms, together called the body. The literals in the body are called *subgoals*. In what follows, we write rules as in the example shown below.

$$\psi \texttt{ :- } [\sim]\phi_1 \texttt{ \& } ... \texttt{ \& } [\sim]\phi_n$$

The following expression is an example of a rule. Here, `r(X)` is the head, the expression `p(X,Y) & q(Y)` is the body; and `p(X,Y)` and `~q(Y)` are subgoals.

```
r(X) :- p(X,Y) & ~q(Y)
```

Intuitively, a rule is something like a reverse implication. It is a statement that the conclusion of the rule is true whenever the conditions are true. For example, the rule above states that `r` is true of any object `X` *if* there is an object `Y` such that `p` is true of `X` and `Y` and `q` is not true of `Y`. For example, if we know that `p(a,b)` is true and `q(b)` is false, then, using this rule, we can conclude that `r(a)` is true. See the end of this section for a more formal treatment of semantics.

A *logic program* is a set of facts and rules of the form just described. Unfortunately, the language of rules, as defined above, allows for logic programs with some unpleasant properties. To eliminate these problems, we concentrate exclusively on logic programs where the rules have two special properties, viz. safety and stratification.

A rule in a logic program is *safe* if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body. A logic program is safe if and only if every rule in the program is safe.

The rule shown below is safe. Every variable in the head and every variable in the negative subgoal appears in a positive subgoal in the body. Note that it is okay for the body to contain variables that do not appear in the head.

```
r(X,Y) :- p(X,Y,Z) & ~q(X,Z)
```

By contrast, the two rules shown below are not safe. The first rule is not safe because the variable `Z` appears in the head but does not appear in any positive subgoal. The second rule is not safe because the variable `Z` appears in a negative subgoal but not in any positive subgoal.

```
s(X,Y,Z) :- p(X,Y)
```

```
t(X,Y) :- p(X,Y) & ~q(Y,Z)
```

(Note that this condition is stronger than necessary. We do not need every rule to be safe; we just require that the program as a whole is safe. The definition of this broader notion of safety is a little complicated and the distinction is unnecessary here, so we skip over this subtlety in the interests of simplicity.)

We say that a set of view definitions is *stratified with respect to negation* if and only if its rules can be partitioned into *strata* in such a way that (1) every stratum contains at least one rule, (2) the rules defining relations that appear in positive goals of a rule appear in the same stratum as that rule *or* in some lower stratum, and (3) the rules defining relations that appear in negative subgoals of a rule occur in some *lower* stratum (not the same stratum).

As an example, assume we have a unary relation `p` that is true of all of the objects in some application area, and assume that `q` is an arbitrary binary relation. Now, consider the ruleset shown below. The first two rules define `r` to be the transitive closure of `q`. The third rule defines `s` to be the complement of the transitive closure.

```
r(X,Y) :- q(X,Y)
r(X,Z) :- q(X,Y) & r(Y,Z)
s(X,Y) :- p(X) & p(Y) & ~r(X,Y)
```

This is a complicated ruleset, yet it is easy to see that it is stratified with respect to negation. The first two rules contain no negations at all, and so we can group them together in our lowest stratum. The third rule has a negated subgoal containing a relation defined in our lowest stratum, and so we put it into a stratum above this one, as shown below. This ruleset satisfies the conditions of our definition and hence it is stratified with respect to negation.

Stratum	Rules
2	`s(X,Y) :- p(X) & p(Y) & ~r(X,Y)`
1	`r(X,Y) :- q(X,Y)` `r(X,Z) :- q(X,Y) & r(Y,Z)`

By comparison, consider the following ruleset. Here, the relation `s` is defined in terms of `p` and the negation of `r`, and the relation `r` is defined in terms of `p` and the negation of `s`.

```
r(X,Y) :- p(X) & p(Y) & q(X,Y)
s(X,Y) :- r(X,Y) & ~s(Y,X)
```

There is no way of dividing the rules of this ruleset into strata in a way that satisfies the definition above. Hence, the ruleset is *not* stratified with respect to negation.

The problem with unstratified rulesets is that there is a potential ambiguity. As an example, consider the rules above and assume that our dataset also included the facts `p(a)`, `p(b)`, `q(a,b)`, and `q(b,a)`. From these facts, we can conclude `r(a,b)` and `r(b,a)` are both true. So far, so good. But what can we say about `s`? If we take `s(a,b)` to be true and `s(b,a)` to be false, then the second rule is satisfied. If we take `s(a,b)` to be false and `s(b,a)` to be true, then the second rule is again satisfied. The upshot is that there is ambiguity about `s`. By concentrating exclusively on logic programs that are stratified with respect to negation, we avoid such ambiguities.

View definitions in static logic programs are required to be both safe and stratified with respect to negation. This is a departure from view definitions in Logic Programming languages like Prolog, which permit rules that are unsafe and logic programs that are not stratified.

The semantics of view definitions in static logic programs can be formalized by defining the result of applying a static logic program to a dataset. The resulting *extension* is the set of all facts that can be "deduced" from the dataset on the basis of the rules in the static logic program.

An *instance* of an expression (atom, literal, or rule) is one in which all variables have been consistently replaced by terms from the Herbrand universe. For example, if we have a language with object constants `a` and `b`, then `r(a) :- p(a,a)`, `r(a) :- p(a,b)`, `r(b) :- p(b,a)`, and `r(b) :- p(b,b)` are all instances of `r(X) :- p(X,Y)`.

Given this notion, we can define the result of single application of a single rule to a dataset. Given a rule r and a dataset Δ, we define $v(r,\Delta)$ to be the set of all ψ such that (1) ψ is the head of an arbitrary instance of r, (2) every positive subgoal in the instance is a member of Δ, and (3) no negative subgoal in the instance is a member of Δ.

Using this notion, we define the result of repeatedly applying the rules in a single stratum Σ to a dataset Δ of facts in the vocabulary of the stratum below. Consider a sequence of datasets defined recursively as follows. $\Gamma_0 = \Delta$, and $\Gamma_{n+1} = \cup v(r,\Gamma_n)$ for all r in Σ. Finally, we define the closure of Σ on Δ to be the union of the datasets in this sequence, i.e. $C(\Sigma,\Delta) = \cup\Gamma_i$.

Finally, we define the *extension* of a static logic program Ω on dataset Δ as follows. Our definition relies on a decomposition of Ω into strata $\Sigma_1, \ldots, \Sigma_n$. Let $\Delta_0 = \Delta$, and let $\Delta_{n+1} = \Delta_n \cup C(\Sigma_{n+1},\Delta_n)$. Since there are only finitely many rules in a static logic program and every stratum must contain at least one rule, there are only finitely many sets to consider (though the sets themselves might be infinite).

It can be shown that there is only one extension for any static logic program applied to any dataset. Although it is sometimes possible to stratify the rules in more than one way, this does not cause any problems. So long as a program is stratified with respect to negation, the definition just given produces the same extension no matter which stratification one uses.

Note that the extension of any function-free static logic program on a finite dataset must be finite. Also, the extension of any non-recursive static logic program applied to a finite dataset must be finite. In both cases, the extension can be computed in time that is polynomial in the size of the dataset.

In the case of recursive programs without function constants, the result must be finite. However, the cost of computing the extension may be exponential in the size of the data, but the result can be computed in finite time.

For recursive programs with function constants, it is possible that the extension is infinite. In such cases, the extension is still well-defined; but in practice it may be necessary to use a different algorithm to compute whether or not a given atom is in the extension. There are multiple ways this can be done. See Ullman's book on Database Systems and Knowledge Base Systems for a discussion of some usable approaches.

4. Operation Definitions

The syntax of operation definitions is analogous to the syntax for view definitions. The various types of constants are the same, and the notions of term and atom and literal are also the same. However, to these, we add a few new items.

To denote operations, we designate some constants as *operation constants*. As with constructors and relation constants, each operation constant has a fixed arity - unary, binary, and so forth.

An *action* is an application of an operation to specific objects. In what follows, we denote actions using a syntax similar to that of atomic sentences, viz. an *n*-ary operation constant followed by *n* terms enclosed in parentheses and separated by commas. For example, if `f` is a binary operation constant and `a` and `b` are symbols, then `f(a,b)` denotes the action of applying the operation `f` to `a` and `b`.

An *operation definition rule* (or, more simply, an *operation rule*) is an expression of the form shown below. Each rule consists of (1) an action expression, (2) a double colon, (3) a literal or a conjunction of literals, (4) a double shafted forward arrow, and (5) a literal or an action expression or a conjunction of literals and action expressions. The action expression to the left of the double colon is called the *head*; the literals to the left of the arrow are called *conditions*; and the literals to its right are called *effects*.

$$\gamma \texttt{ :: } [\sim]\phi_1 \texttt{ \& } \ldots \texttt{ \& } [\sim]\phi_m \texttt{ ==> } [\sim]\psi_1 \texttt{ \& } \ldots \texttt{ \& } [\sim]\psi_n \texttt{ \& } \gamma_1 \texttt{ \& } \ldots \texttt{ \& } \gamma_k$$

Intuitively, the meaning of an operation rule is simple. If the conditions of a rule are true in any state, then executing the action in the head requites that we execute the effects of the rule.

For example, the following rule states that in any state in which `p(a,b)` is true and `q(a)` is false, the executing `click(a)` requires that we remove `p(a,b)` from our dataset, add `q(b)`, perform action `click(b)`.

```
click(a) :: p(a,b) & ~q(a) ==> ~p(a,b) & q(a) & click(b)
```

As with rules defining views, operation rules may contain variables to express information in a compact form. For example, we can write the following rule to generalize the preceding rule to all objects.

```
click(X) :: p(X,Y) & ~q(X) ==> ~p(X,Y) & q(X) & click(Y)
```

As with view rules, *safety* is a consideration. Safety in this case means that every variable among the effects of a rule or in negative conditions also appears in the head of the rule or in the positive conditions.

The operation rules shown above are both safe. However, the rules shown below are not. The second effect of the first rule contains a variable that does not appear in the head or in any positive condition. In the second rule, there is a variable that appears in a negative condition that does not appear in the head or in any positive condition.

```
click(X) :: p(X,Y) & ~q(X) ==> ~p(X,Y) & q(Z) & click(Y)
click(X) :: p(X,Y) & ~q(Z) ==> ~p(X,Y) & q(X) & click(Y)
```

In some operation rules there is no condition, i.e. the effects of the transition rule take place on all datasets. We can, of course, write such rules by using the condition `true`, as in the following

example.

```
click(X) :: true ==> ~p(X) & q(X)
```

For the sake of simplicity in writing our examples, we sometimes abbreviate such rules by dropping the conditions and the transition operator and instead write just the effects of the transition as the body of the operation rule. For example, we can abbreviate the rule above as shown below.

```
click(X) :: ~p(X) & q(X)
```

An *operation definition* is a collection of operation rules in which the same operation appears in the head of every rule. As with view definitions, we are interested primarily in rulesets that are finite. However, in analyzing operation definitions, we occasionally talk about the set of all ground instances of the rules, and in some cases these sets are infinite.

The semantics of operation definitions is more complicated than the semantics of updates due to the possible occurrence of views in the conditions of the rule and the possible occurrence of operations in its effects. In what follows, we first define the expansion of an action in the context of a given dataset, and we then define the result of performing that action on that dataset.

Suppose we are given a set Ω of rules, a set Γ of actions (factoids, negated factoids, and actions), and a dataset Δ. We say that an *instance* of a rule in Ω is *active* with respect to Γ and Δ if and only if the head of the rule is in Γ and the conditions of the rule are all true in Δ.

Given this notion, we define the *expansion* of action γ with respect to rule set Ω and dataset Δ as follows. Let Γ_0 be $\{\gamma\}$ and let Γ_{i+1} be the set of all effects in any instance of any rule in Ω with respect to Γ_i and Δ. We define our expansion $U(\gamma,\Omega,\Delta)$ as the fixpoint of this series. Equivalently, it is the union of the sets Γ_i, for all non-negative integers i.

Next, we define the positive updates $A(\gamma,\Omega,\Delta)$ to be the positive base factoids in $U(\gamma,\Omega,\Delta)$. We define the negative updates $D(\gamma,\Omega,\Delta)$ to be the set of all negative factoids in $U(\gamma,\Omega,\Delta)$.

Finally, we define the result of applying an action γ to a dataset Δ as the result of removing the negative updates from Δ and adding the positive updates, i.e. the result is $(\Delta - D(\gamma,\Omega,\Delta)) \cup A(\gamma,\Omega,\Delta)$.

To illustrate these definitions, consider an application with a dataset representing a directed acyclic graph. In the sentences below, we use symbols to designate the nodes of the graph, and we use the `edge` relation to designate the arcs of the graph.

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

The following operation definition defines a ternary operation `copy` that copies the outgoing arcs in the graph from its first argument to its second argument.

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

Given this operation definition and the dataset shown above, the expansion of `copy(b,c)` consists of the changes shown below. In this case, the factoids representing the two arcs emanating from `b` are all copied to `c`.

```
edge(c,d)
edge(c,e)
```

After executing this event, we end up with the following dataset.

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

The following rule defines a unary operation `invert` that reverses the incoming arcs of the node specified as it argument.

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

The expansion of `invert(c)` is shown below. In this case, the arguments in the factoid with `c` as second argument have all been reversed.

```
~edge(c,d)
~edge(c,e)
edge(d,c)
edge(e,c)
```

After executing this event, we end up with the following dataset.

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

Finally, the following operation rules define a binary operation that inserts a new node into the graph (the first argument) with an arc to the second argument and arcs to all of the nodes that are reachable from the second argument.

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

The expansion of `insert(w,b)` is shown below. The first rule adds `edge(w,b)` to the expansion. The second rule adds `insert(w,d)` and `insert(w,e)`. Given these events, on the next round of expansion, the first rule adds `edge(w,d)` and `edge(w,e)` and the second rules adds `insert(w,c)`. On the third round of expansion, we get `edge(w,c)`. At this point, neither rule adds any additional items to our expansion, and the process terminates.

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

Applying this event to the preceding dataset produces the result shown below.

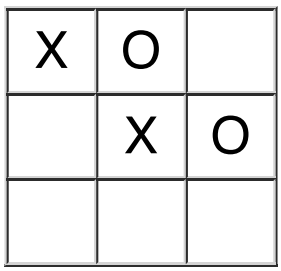
```
edge(a,b)
edge(b,d)
edge(b,e)
edge(d,c)
edge(e,c)
edge(w,b)
edge(w,d)
edge(w,e)
edge(w,c)
```

Note that it is possible to define `insert` in other ways. We could, for example, define a view of `edge` that relates each node to every node that can be reached from the node; and we could then use this view in a non-recursive definition of `insert`. However, this would require us to introduce a new view into our vocabulary; and, for many people, this is less clear than the definition shown above.

5. Example - Tic Tac Toe

As an example of a dynamic logic program, 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. The figure below shows one state of play in Tic Tac Toe.



In our definition of Tic Tac Toe, states are characterized by the contents of the cells on the Tic Tac Toe board and control (whose turn it is to play). (It is true that control can be defined in terms of the contents of cells; but making control explicit costs little and simplifies the description.) 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. The dataset shown below uses this vocabulary to characterize the game state show above.

```
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)
```

Our first step is to define legality of moves. A player may mark a cell if that cell is blank. Otherwise, it has no legal actions.

```
legal(M,N) :- cell(M,N,b)
```

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

```
mark(M,N) :: control(Z) ==> ~cell(M,N,b) & cell(M,N,Z)
mark(M,N) :: control(x) ==> ~control(x) & control(o)
mark(M,N) :: control(o) ==> ~control(o) & 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.

```
row(M,Z) :- cell(M,1,Z) & cell(M,2,Z) & cell(M,3,Z)
column(M,Z) :- cell(1,N,Z) & cell(2,N,Z) & cell(3,N,Z)
diagonal(Z) :- cell(1,1,Z) & cell(2,2,Z) & cell(3,3,Z)
diagonal(Z) :- cell(1,3,Z) & cell(2,2,Z) & cell(3,1,Z)
```

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

```
line(Z) :- row(M,Z)
line(Z) :- column(M,Z)
line(Z) :- diagonal(Z)
```

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

```
terminal :- line(x)
terminal :- line(o)
terminal :- ~open
```

```
open :- cell(M,N,b)
```

Our rules specify the states and physics of the game. They do not specify how to play the game effectively. In order to decide this, a player needs to consider the effects of his 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.

6. Comparison to Other Languages

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 this full paper, we summarize these approaches and highlight their commonalties and differences. We mention just two of these below.

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.

7. Conclusion

In practice, it is common to extend the simple version of Dynamic Logic Programming described here to include "built-in" relations (e.g. arithmetic) and other operators (e.g. aggregates). The syntax and semantics of such extensions are a little messy. Luckily, they pose no significant theoretical challenges; and, in the interest of brevity, they are not covered here.

The intent of this article is to provide a concise but reasonably rigorous account of the syntax and semantics of Dynamic Logic Programming. For motivation and examples of all of these concepts, see the textbook Dynamic Logic Programming.

References

W. F. Clocksin, and C. S. Mellish: Programming in Prolog. 4th edition. New York: Springer-Verlag. 1994.

A. J. Bonner, M. Kifer: Transaction Logic Programming, International Conference on Logic Programming (ICLP), 1993.

D. Cabeza, M. Hermenegildo: Distributed www programming us- ing (ciao-)prolog and the pillow library. Theory and Practice of Logic Programming, 1(3):251-282, May 2001.

S. Constantini, A. Tocchio: The DALI Logic Programming Agent-Oriented Language. Alferes, J.J., Leite, J. (eds) Logics in Artificial Intelligence. JELIA 2004. Lecture Notes in Computer Science, vol 3229. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30227-8_57

S. Flesca, S. Greco: Declarative Semantics For Active Rules. Theory and Practice of Logic Programming, 1(1): 43-69, 2001.

P Fodor: Practical Reasoning with Transaction Logic Programming for Knowledge Base Dynamics, PhD Thesis, Stonybrook University.

T. Fruehwirth: Constraint Handling Rules. Cambridge University Press, ISBN 9780521877763, 2009.

M. Genesereth, N. Love, B. Pell: The International Game Playing Competition. AAAI Magazine, 2005.

M. Genesereth: Dynamic Logic Programming.
http://logicprogramming.stanford.edu/miscellaneous/dlp.html

M. Genesereth: Epilog. http://epilog.stanford.edu

M. Genesereth, V. Chaudhri: Logic Programming. Synthesis Lectures on Artificial Intelligence and Machine Learning, February 2020. https://doi.org/10.2200/S00966ED1V01Y201911AIM044

P. Hayes: Computation and deduction. Proceedings Second Symposium on Mathematical Foundations of Computer Science, Czechoslovakian Academy of Sciences, Czechoslovakia, 1973, pp. 105-118.

M. Kifer, A. Liu: Declarative Logic Programming, ACM Books, 2018.

R.Kowalski: Algorithm = Logic + Control. Communications of the ACM, July 1979, Vol 22 No 7.

R. Kowalski, F. Sadri: LPS-A Logic-based Production System Framework. 2009.

R. Kowalski, F. Sadri: Integrating Logic Programming and Production Systems in Abductive Logic Programming Agents. 2009.

M. Slota, J. A. Leite: EVOLP: An Implementation. Computational Logic in Multi-Agent Systems, 8th International Workshop, CLIMA VIII, Porto, Portugal, September 10-11, 2007.

D. S. Warren: Programming in Tabled Prolog. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.4635

N.-F. Zhou: The Language Features and Architecture of B-Prolog. Theory and Practice of Logic Programming 12(1-2), 2011. DOI: 10.1017/S1471068411000445