Basic Model:

Here we will introduce some of the basic terminology and formula of PEL and PEL inference.

The first thing that any PEL application must specify is the set of event symbols that can be used. An event symbol is just a string that gives a name of a particular event of interest. For example,

Defense-P1 and Shooting-P1 and HasBall-P2

are two event symbols that might be used in our basketball application, with the intended meaning of "Player 1 is on defense", "Player 1 is shotting", and "Player 2 has the ball". Generally the event symbols will be divided into observable and hidden events. For example in our basketball application the detectors yield observable event symbols such as D-Shooting-P1, while the corresponding hidden event is Shooting-P1.

An event symbol by itself does not have a truth value. Rather, truth values are assigned to event occurences. An "event occurence" is of the form :

E@I

where E is an event symbol and I is a time interval of the form [a,b] where a and b are natural numbers. Asserting that E@I is true means that an instance of event E occured precisely over interval I, or in other words E was coincident with I. We will also allow event occurences to be specified via spanning intervals, to allow for compact encodings. Thus if E is an event symbol and S is a spanning interval, then asserting that

E@S

is true indicates that E@I is true for each I \in S.

We now need to introduce the notion of an "interpretation". An interpretation over a set of event symbols {E1,...,En} is a set of event occurences over those event symbols.

Thus an example of an interpretation is M={E1@S1, E4@S4} indicating that E1 occurse at each I \in S1 and E4 occurs at each I in S4. Note that if an event symbol does not appear in an interpretation then it is assumed to not occur at any interval. So, for example, E2 does not occur anywhere in the interpretation M. Note that any interpretation will have a maximum time point that it mentions and we will take this to be the maximum time allowed in the interpretation and will generally refer to this time as T.

Given a video V, our basic problem is to compute an interpretation M that accurately reflects the event occurences in V. Typically this will be done as follows. First, a set of event detectors will be run on V which produce a set of observed event occurences {O1@S1, ..., Ok@Sk} where the Oi are observable event symbols and Oi@Si indicates that the detector asserts that Oi occurred at each I in Si. In basketball, the detector might produce an event occurence such as

D-Dribbling-P1@[[1,10],[1,10]]

indicating that player 1 was dribbling the ball along all sub-intervals of [1,10]. Note that it is not necessarily the case that the player was dribbling in reality. Rather this provides evidence of dribbling and the actual act of dribbling must be inferred.

Every interpretation of V will contain the observed event occurences. The goal now is the "complete the interpretation" by inferring the event occurences involving the hidden event symbols. For example, if we had two observed event occurences:

D-Dribbling-P1@[[1,4],[1,4]] and
D-Dribbling-P2@[[3,6],[3,6]]

then a reasonable full interpretation might be:

{D-Dribbling-P1@[[1,4],[1,4]],D-Dribbling-P2@[[3,6],[3,6]]
Dribbling-P1@[[1,3],[1,3]], Dribbling-P2@[[4,6],[4,6]]}

which enforces a constraint that no two players can be dribbling the ball at the same time, despite the fact that the detected events allowed for this.

The question is on what basis should we infer the hidden event occurences. We will do this by specifying constraints (via PEL), both hard and soft, that are used to score the goodness of an interpretation. The highest scoring interpretation will then be output. This is identical to CRF literature where on is given a set of X variables and one must infer the values of a set of hidden Y variables. In our case, X corresponds to the observed event occurences and Y corresponds to the hiddent event occurrences. So it remains to specify the syntax of PEL constraints and how they are used to score interpretations.

I won't write out the full syntax of event logic formulas here, though some detailed syntactic restrictions are mentioned below in the inference sections. Rather I'll assume that one knows about event logic and focus on the extension to PEL.

A PEL knowledge base (KB) is a set of weighted event logic formulas:

{(phi1,w1),(phi2,w2),...,(phim,wm)}

where the Fi are event logic formulas and the Wi are numeric weights (very large values will effectively act like hard constraints).

An event logic formula phi by itself does not have a meaning (or value) unless it is in the context of an interpretation. (Much like a CRF potential doesn't have a value without an assignment to X,Y).

Given an interpretation M and a formula phi we can ask for the set of intervals in M (i.e. intervals in [0,T]) where phi is true/satisfied. We will denote this set as:

SAT(M,phi)

which we will general represent in terms of a set of spanning intervals. We will sometimes use the notation phi@S to indicate that phi is true along all I in spanning interval S in M.

So for example, given the interpretation

M= {D-Dribbling-P1@[[1,4],[1,4]],D-Dribbling-P2@[[3,6],[3,6]]
Dribbling-P1@[[1,3],[1,3]], Dribbling-P2@[[4,6],[4,6]]}

and the simple primitive formula,

phi = Dribbling-P1

we have that

SAT(M,phi) = [[1,3],[1,3]]

similarly if we have

phi = D-Dribbling-P1 & D-Dribbling-P2

we get that

SAT(M,phi) = [[3,4],[3,4]]

Now given an interpretation M and a PEL KB Sigma = {(phi_1,w_1),...,(phi_m,w_m)} We can define the score of M wrt Sigma:

SCORE(M,Sigma) = sum_i w_i*|SAT(M,phi_i)|

where |SAT(M,phi)| is the number of intervals in SAT(M,phi). That is the score of an interpretation is just the weight of each formula multiplied by the number of intervals it is true over. If a weight is very large then the corresponding formula will be treated as a hard constraint that must always be satisfied along all intervals of a model.

To summarize the important terms to get straight are "event symbol", "event occurence", "interpretations", "PEL formulas", "PEL KB", "SAT", and "SCORE".

Below we cover some details of he event symbols and formulas that appear useful for basketball and then go into the inference problems: computing SAT and maximizing the score.

Primitive Event Symbols:

Here is the list of primitive detector event symbols. Rather than write out a symbol for each player we use templates such as E(?p) which should be thought of as a set of event symbols, one for each player, e.g. E-P1, E-P2, E-P3, E-P4.

1) D-Bouncing
2) D-BallTrajectory
3) D-Dribbling(?p)
4) D-Jumping(?p)
5) D-Shooting(?p)
6) D-Passing(?p)

7) D-Catching(?p)
8) D-CloseToRim
9) D-MovingTogether(?p1,?p2)
10) D-HasBall(?p)

The detector events will be treated as liquid.
After applying the detectors we will get a set of detected events. E.g.

D-Passing(a)@[1:10]
D-Shooting(a)@[10:20]

where 'a' is a player object. The detections will be prepresented in terms of spanning intervals.
Above [a:b] is shorthand for the spanning interval [[a,b],[a,b]].

Here are the semantic/hidden event symbols that we will work with. I'll start with the events that we are most interested in infering.
Some of these are associated with detectors and some are not.

Note that there is going to be a semantic event corresponding to nearly all of the detector events, which should be viewed as the filtered version of the detector. I'll start with the events that we are most interested in infering.

1) PassTo(?p1,?p2) ;; this could either be coincident with Passing or non-liquid beginning with passing and ending with catching
2) Passing(?p)
3) Catching(?p)
4) Shooting(?p)
5) HasBall(?p)
6) Dribbling(?p)
7) HoldingBall(?p) ;; maybe
8) Defense(?p)
9) Offense(?p)
10) Defending(?p1,?p2)
11) Rebound(?p1) ;; maybe

The remainder of the semantic predicates correspond to the remaining detectors, though it is not clear that we will actually need to use all of these, by convention we will include them.

12) Bouncing ;; maybe unnecessary
13) BallTrajectory
14) Jumping(?p)
15) CloseToRim ;; maybe unnecessary
16) MovingTogether(?p1,?p2) ;; maybe unnecessary

All of the above events are liquid, with the possible exception of PassTo

---

Formulas:

We will relate the detector events to the semantic primitive events via weighted implications:

D-Shooting(?p) -> Shooting(?p)
D-Passing(?p) -> Passing(?p)
D-BallTrajectory -> BallTrajectory

Again the above we have used template variables for shorthand. Really the first formula would have 4 versions, one for each player, e.g.

D-Shooting-P1 -> Shooting-P1

The weights for the above formulas are not shown, but the magnitude of the weight will reflect our confidence in a detector. For example, given a large weight for the the first formula and an observation:

D-Shooting-P1@[5:10]

there would be a high cost to not including Shooting-P1@[5:10] in the interpretation, since not doing so would violate the implication.

We may need to also include the reverse implications, but not clear yet. E.g. Shooting(x) -> D-Shooting(x). All of these formulas are
liquid constraints---meaning that if they are true/false over an interval, they are true/false over all sub-intervals.

We will also have some formulas that just relate the semantic (non-motion) events. These formulas constitute the domain knowledge that has the role of filtering the detectors and making higher-level inferences.

Here are some formulas that seem useful---not a complete list. I've tried to group these formulas into specific classes of formulas according to their syntactic structure. Below Ex1 (short for ExactlyOne), which is a formula that is true if exactly one of the arguments is true. For example Ex1(p,q,r) is true when exactly one of p, q, or r are true (same as xor when there are just 2 arguments).

We will sometimes use Ex1 as a variable quantifier, so if x is a variable over the the possible objects {a,b} and E is an event formula, then:

Ex1 x E(x) = Ex1(E(a),E(b)).

Similarly we use the notation AM1 for atmost one.

We will use \D_{r} to represent the diamond notation in event logic.

*Formula Type 1: Liquid Formulas

Ex1(Defense(x),Offense(x))
[Defending(x,y) -> (Defense(x) & Offense(y))]
[Defending(x,y) -> MovingTogether(x,y)]
[Defending(x,y) <- (Defense(x) & Offense(y) & MovingTogether(x,y))]
[PassTo(x,y) -> (Offense(x) & Offense(y))]
[Shooting(x) -> Offense(x)]

[HasBall(x) -> Ex1(Dribble(x),HoldingBall(x),Shooting(x),Passing(x))] ;; convention that a player hasball when shooting and passing
[Dribble(x) -> HasBall(x)]
[HoldingBall(x) -> HasBall(x)]
[Shooting(x) -> HasBall(x)]
[Passing(x) -> HasBall(x)]
[HasBall(x) -> !BallTrajectory]
[AM1 x HasBall(x)] ;; AM1 reads "at most 1"
[Dribbling(x) -> Bouncing]
[Bounching -> Dribbling(x)]

*Formula Type 2: [A1 & ... & An] -> \D_{r} [E1 or ... or Ek] for r \in {m,mi,fi,f] where Ai are primitives and Ei are primitives or negations of primitives

Shooting(x) -> \D_{mi} [Shooting(x) or BallTrajectory]
Passing(x) -> \D_{mi} [Passing(x) or BallTrajectory]
Catching(x) -> \D_{mi} [Catching(x) or HasBall(x)]
Catching(x) -> \D_{m} [Catching(x) or !HasBall(x)]
[HasBall(x) & Jumping(x)] -> \D_{mi} [Jumping(x) or ShootBall(x)]
[HassBall(x) & Jumping(x)] -> \D_{mi} [Jumping(x) or !HasBall(x)] ;; can't jump with ball and then land with ball (up-and-down)

HasBall(x) -> \D_{fi} [HasBall(x) or Passing(x) or Shooting(x)]

*Formula Type 3: E -> \D_{mi} (E or (E1 ; ... ; Ek))

Shooting(x) -> \D_{mi} (Shooting(x) or (BallTrajectory ; NearRim))

*Formulat Type 3: [E1 ; ... ; Ek] -> \D_{m} [A1 or ... or An]

[BallTrajectory ; NearRim] -> \D_{m} [BallTrajectory or (Ex1 x Shooting(x))]

[BallTrajectory ; Catching(x)] -> \D_{m} [BallTrajectory or (Ex1 x Passing(x))]

*Formula Type 4: E -> \D_{mi} (E or (E1 ; ... ; Ek))

Shooting(x) -> \D_{mi} (Shooting(x) or (BallTrajectory ; NearRim))

*Formula Type 5: E -> ![E1 ; E2 ; ... ; Ek]

HasBall(?x) -> ![Dribbling(?x) ; !Dribbling(?x) ; Dribbling(?x)]

![HasBall(?x) ; Jump(?x) ; HasBall(?x)]

*Formula Type 6: E -> [E1 ; E2 ; ... ; Ek] ;; note that here E will be non-liquid

PassTo(?x,?y) -> [Passing(?x) ; BallTrajectory ; Catching(?x)]

PassTo(?x) -> [HasBall(?x) ; BallTrajectory ; HasBall(?y)]

*Formula Type 7: E <- [E1 ; E2 ; ... ; Ek]

PassTo(?x,?y) <- [Passing(?x) ; BallTrajectory ; Catching(?x)]

Local Search Inference:

Initialize interpretation M(0); i=0

Repeat for N iterations

  Select a violated formula phi at random

  Compute spanning interval S = SAT(M(i),!phi)

  Moves = set of interpretation changes that satisfy or partially satisfy phi@S
  ;; key step to define

  With probability p : M(i+1) = apply random move
            else : M(i+1) = select best move

  i++

Return highest scoring Mi

Below we first describe the process of computing SAT. Next we describe the approach for generating search moves.

Computing SAT(M,phi):

First a note about the variables in the above formulas. Most formulas include variables such as the ?x in:

    Defense(?x) or Offense(?x)

For now we are simply treating these variables as defining a formula template that will be expanded to a set of "ground formulas" after a set of objects are known. Lets say that each variable can have a type (possibly the universal type). In our domain the types will be players and ball. Given a particular video, we are currently assuming that we have extracted a set of objects, each with an associated type. In our video there will be 4 player objects say {p1,p2,p3,p4} and 1 ball object b. Given these objects, we will turn the formula templates into ground formulas by instantiating the variables in all possible ways that are type consistent. So above we would get a set of four formulas, one for each way of plugging in a player for ?x.

Each formula tempplate will also be associated with a weight, which could be infinity. The ground formulas will simply inherit that weight.

The description below is for evaluating SAT on ground formulas. Thus, the discussion below assumes that the instantiating of templates has already beeen done.

Liquid Event Formulas:

Suppose that phi is liquid, meaning that it is composed of only liquid properties using purely logical operations (no temporal operators).

We assume that the primitive liquid event occurences in M are represented via spanning intervals. Note that for a liquid event L the spanning interval representation can be viewed as a disjoint set of segments of the time line. For example, L@{[1:10],[15:20]} indicates that L occured during all intervals within 1:10 and 15:20.

Note that here [i:j] is shorthand for [[i,j],[i,j]]. I think that it will be worth using this shorthand within the system and explicitly distinguishing between spanning intervals that are liquid and those that are not, which use the more general representation involving 4 numbers.

It will be useful to define a few operations specialized to liquid spanning intervals.

Cluster: Given a set of liquid spanning intervals {S1,...,Sk} we define Cluster(S1,...,Sk) to be the successive merging of spanning intervals that overlap or meet until all spanning intervals are disjoint. (Should come up with a semantic description of this). This operation is easy. All one needs to do is merge pairs of spanning intervals that overlap. E.g.

Cluster([1:3],[2:4],[6:7]}) = {[1:4],[6:7]}

note that if the final spanning interval had been [5:6] instead of [6:7] we would have had

Cluster([1:3],[2:4],[5:7]}) = {[1:7]}

since [2:4] meets [5:7].

We will also want the function GAP(S1,...,Sk,T) for a set of liquid spanning intervals and maximum time horizon T, which simply returns part of the time line not covered by S1,...,Sk (i.e. the gaps).

So we have

GAPS([1:3],[5:6],[10:15],15) = {[4:4],[7:9]}

Now we can define how to compute a compact liquid spanning interval representation of SAT(M,phi) for any liquid event formula phi. We will consider the cases:

1) phi = !phi1

SAT(M,phi) = GAPS(SAT(M,phi1))

2) phi = phi1 or phi2

SAT(M,phi) = Cluster(SAT(M,phi1),SAT(M,phi2))

One could define conjunction explicitly or via ! and or.

I think that we will also be using the non-primitive logical operator Ex1 frequently. This can be expanded in terms of primitive operators, e.g.

Ex1(p1,p2,p3) <-> (p1 or p2 or p3) & (!p1 or !p2) & (!p1 or !p3) & (!p2 or !p3)

however, there is probably a more efficient way of computing the spanning intervals of Ex1 by directly processing the spanning intervals of the arguments.

Non-Liquid Formulas:

Non-liquid formulas are formed unary and binary temporal operators applied to sub-formulas. The simplest sub-formulas are primitive events. We will also treat liquid formulas as a base case in this section, since we have shown how to compute their SAT above. Currently the set of temporal operators used above are quite limited. These include:

The unary operators \D_{r} for r \in {m,mi,fi}.

The binary operator "meet" denoted by ";".

The negation "!" operator and conjuction disjunction, from which we can get implication.

Computing SAT(M,phi) for a non-liquid formula is simply a recursive process, the steps of which we outline below for the operators used above.

Consider first

SAT(M,\D_{r} phi)

For a spanning interval S, define D(r,S) to be the set of all intervals I such that (J r I) for some J \in S. Thus, we get that

SAT(M,\D_{r} phi) = D(r,SAT(M,phi))

For our specific cases of r \in {mi,m,f} we get:

D(mi,[[a,b],[c,d]]) = [[0,T],[a,b]] ;; assuming there are really intervals that begin with a and b, which should be the case for normalized spanning intervals.

If Phi is represented as a set/union of spanning intervals then we simply get the union of the individual intervals desribed above.

Simiarly we get:

D(m,[[a,b],[c,d]]) = [[c,d],[0,T]]

Also we get:

D(f,[[a,b],[c,d]]) = [[0,b-1],[c,d]]

Note that even if phi is liquid \D_{r} phi will typically not be liquid.

The only binary operator that we current have is meets. Futhermore, in our current formulas this operator is applied in a restricted way. In particular, the operater is always applied to a sequence of liquid events (or liquid formulas). That is, the meets operator always appears in the form (E1 ; E2 ; ... ; Ek) where the Ei are liquid events/formulas. This suggests that we should handle this case in a specialized way. (Should verify that this specialized handling is more efficient than handling the general case.)

This seems pretty straightforward given the liquid spanning intervals for the Ei. Recall that this representation can be viewed as a disjoint set of sub-intervals of [0,T].

First compute the spanning interval for SAT(M, E1;E2). This is the set of intervals equal to span(I,J) for some I \in SAT(M,E1) and J \in SAT(M,E2).
Let SAT(M,E1) = [a1:b1] and SAT(M,E2) = [a2:b2]. If these spanning intervals do not overlap then we end up with an empy set of intervals. Otherwise if they
overlap then we end up with:

SAT(M,E1;E2) = [[a1:b1],[a2,b2]]

In general if E1 and E2 were sets of spanning intervals then we would have a set for SAT(M,E1;E2) with an element for each pair of intersecting spanning intervals.

We can then proceed to process E3. If it intersects with [a2,b2] then we will get:

SAT(M,E1;E2;E3) = [[a1:b1],[a3,b3]].

In general we can proceed in this way as long as the Ei intersects E_{i-1}. All of the different ways of extending to Ei results in a multiplicative blowup in the number of spanning intervals.
One way to view this process is as a layered graph search. Level i has a node for each spanning interval in SAT(M,Ei). There is an edge between a node in layer i and layer i-1 if the corresponding spanning intervals overlap. Now if there is a path from node S in layer 1 to node S' in layer k, where S = [[a1,b1],[c1,d1]] and S'=[[ak,bk],[ck,dk]] denote the corresponding spanning intervals of the nodes we can add [[a1,b1],[ck,dk]] to the spanning interval set of SAT(M,E1;...;Ek). This view as a graph search might suggest a very efficient implementation for our special case.

I haven't thought about whether a similar approach works for meets of non-liquid events.

Now that we can process the \D and meets operator all that is left is to combine these results via the logical operators. This is straightforward using the basic operations on spanning intervals.

For disjunction we get:

SAT(M,E1 or E2) = {SAT(M,E1),SAT(M,E2)}

For conjunction we get:

SAT(M,E1 & E2) = INTERSECT(SAT(M,E1),SAT(M,E2))

For negation !E of a possibly non-liquid event formula E we simply get the complement of SAT(M,E). The complement of a single spanning interval is given by:

NOT([[a,b],[c,d]]) = {[[0,a-1],[0,T]],[[b+1,T],[0,T]],[[0,T],[0,c-1]],[[0,T],[d+1,T]]

giving 4 intervals.

If SAT(M,E) is a set {S1,...,Sk} then we can get:

SAT(M,!E) = INTERSECTION_i NOT(Si)

We use implication above E1 -> E2 which is equivalent to (!E1 or E2) and thus can be computed using the above operations.

It should now be possible to compute SAT(M,E) for any formula that fits the form assumed above. We can add machinery to handle other formula constructs as needed. However, implementing a fully general evaluator is not worth the time before the CVPR deadline.

Search Moves:

Given a interpretation M and a formula E we can compute SAT(M,!E) to get the intervals where it is violated. We now need to define a set of moves that can correct some or all of the violations. At this point we will focus on defining specialized moves for each of our formula types. At this point we will focus on specifying an intuitive set of moves that we believe will work well for our data.

*Liquid Formulas:

We first notice that for our current formula set we can restrict our attention to a fragment of all possible liquid formulas. The space of all possible liquid formulas corresponds to the space of all possible propositional logic formulas and if left unrestricted we will end up needing to solve general SAT problems in our inference process. This is certainly feasible, but requires more machinery than we want to implement at this time. Here is the syntax for liquid formulas that we will allow:

A literal L is either a liquid primitive event symbol E or its negation !E (e.g. Defense(p1) and !Defense(p1) are literals).

A literal L is a legal liquid formula.

A conjunction L1 & ... & Ln of n consistent literals is a legal liquid formula

EX1(L1,...,Ln) is a legal liquid formula for literals Li

AM1(L1,...,Ln) is a legal liquid formula for literals Li

A disjunction F1 or ... or Fn of legal liquid formulas Fi is a legal liquid formula

Notice that we rule out conjunctions over complex structures in the above, which is what would complicate defining our local stochastic search procedure. All of the example liquid formulas described above can be written in this syntax noting that A -> B is equivalent to !A or B.

Suppose now that during our stochastic local search procedure that the current interpretation is M and we select a violated liquid formula phi. Let,

SAT(M,[!phi]) = {S1,...,Sn}

be the set of disjoint liquid spanning intervals over which phi is always false. (This is different than ![phi], which would be the intervals where E is not true throughout.)

We want to define a set of moves (each a local change to M) that partially satisfy phi. For now we will define a set of moves for each of the Si and the union of all these moves will be the ones we consider (or perhaps each search step just randomly selects a single Si). This allows us to focus on a single Si. At this point we will consider moves that will satisfy phi throughout all of Si. Later we may consider moves that satisfy phi over just a window of Si if needed. Thus we want to define the function MOVES(M,Si,phi) which will return a set of interpretations derived from M that satisfy phi throughout Si. We will define this function by induction on the structure of phi. First we need some notation for describing moves.

All moves in MOVES(M,Si,phi) will be defined by starting from M and adding and deleting event occurences over Si. So an example move might be:

{Add(E@Si),Del(E'@Si)}

which if applied to M would produces a interpretation that is identical to M, except that the fact E@Si has been inserted and E'@Si has been removed, which means that if E' had been true along any part of Si in M those facts are removed.

Given a potential move K we can also define the negation NEG(K) to be identical to K except that the Adds become Dels and the Dels become Adds. We can now define MOVES(M,Si,phi) (we'll use shorthand Moves(phi)):

1) phi = E, where E is positive literal
Moves(phi) = {{Add(E@Si)}} ;; note that Moves(phi) is a set of moves, where each move is a set of adds and dels

2) phi = !E, where !E is a negative literal
Moves(phi) = {{Del(E@Si)}}

3) phi = L1 & ... & Ln, where Li are literals
Moves(phi) = {Add(E@Si) | for each positive Li = E} + {Del(E@Si) | for each negative Li = !E}

4) phi = EX1(L1,...,Ln)
Moves(phi) = union_i Moves(Li & conj_{j \not= i} !Lj)

5) phi = AM1(L1,...,Ln)
Moves(phi) = Moves(EX1(L1,...,Ln)) + Moves(!L1 & !L2 & ... & !Ln)

6) phi = F1 or ... or Fn

Moves(phi) = union_i Moves(Fi)

This allows for one to recursively compute a set of moves that will satisfy phi along Si. Each of these moves can be scored according to the score of the resulting interpretation wrt all formulas and the best move selected with some probability, and a random move otherwise.

There is some question about the innefficiency of rescoring each interpretation that results from a move from scratch. Many formulas will not be impacted by a local move and do not need to be re-evaluated (compute SAT for them), but this would require some extra book keeping. I suggest that we save this for later and consider it if this source of innefficiency becomes an issue.

I believe that for the formulas we allow, the set of moves will be complete for satisfying phi along all of Si. However, the overall completeness of the local search under these moves is still a question. I suspect that completeness might require considering moves in only windows of the Si. But if we find that is necessary we can easily allow for that by restricting the above moves to sub-windows of the Si. Doing that would definately provide completeness with respect to liquid formulas.

---

*Formula Type 2: phi1 -> \D_{r} (phi1 | phi2), where r \in {m,mi,fi,f] where phi1 is a conjunction or disjunction of literals and phi2 is a liquid formula using above syntax

Lets first consider the case when r = mi.

Given the above structure we can compute SAT(M,!phi) = {S1,...,Sn}. The Si in this case will not be liquid, but they will all have the same form, in particular, Si=[[ai,bi],[bi,bi]] where phi1@[ai:bi] is true in M and [!phi2]@[bi+1:bi+1] is also true. That is, bi is the pint where phi1 ends and where phi2 is not true at the next point. We will refer to the bi as the critical points. Note that it is probably not necessary to run the general SAT routine to compute the critical points {b1,...,bn}. Rather given the spanning intervals for phi1 and phi2 the critical points can be easily derived from them using a specialized routine.

For each of the critical points bi we can define thre high-level moves:

1) Extending the extent of phi1 in M to satisfy the violation at bi.

To do this let t' be the first point greater than bi where phi2 is true in M. If there is not such point then (1) generates no moves.The corresponding moves are:

Moves(M,[bi+1:t'-1],phi1)

Since phi1 is a simple liquid formula we can use the "moves" procedure described above to compute this set of moves.

Pictorially this move goes form a model that looks like:

```
      2222222
11111
```

to one that looks like:

```
      2222222
11111111
```

2) Shrinking the extent of phi1 in M to satify the violation at bi

Let t' be the latest point in [ai,bi] where phi2 is true or 0 if there is no such point. The moves for this option are:

Moves(M,[t'-1:bi],[!phi1])

Note that here !phi is either a simple conjunction or disjunction of literals since phi is either a conjunction or disjunction of literals.

Pictorially this move goes form a model that looks like:

```
    2222222
11111111111111111111
```

to one that looks like:

```
    2222222
11111111111
```

3) Asserting that phi2 is true for some spanning interval starting at bi + 1

To do this let t' be the first point greater than bi where phi2 is true in M if any or be bi+1 if there is no such point. The corresponding moves are:

Moves(M,[bi+1:t'-1],phi2)

Since phi2 is assumed to fall in our syntactically restricted space we can use the above procedure to compute the moves.

Pictorially this move goes form a model that looks like:

```
      2222222
11111
```

to one that looks like:

```
    2222222222
11111
```

So the full set of moves is the union across all bi of the three sets of moves defined for each bi. Again here there is the choice of evaluating them all and picking the greedy one, or just a subset of them.

We can work out a similar set of moves for the case when r=f. I'll write out the details of this, noting that fi and m are similar:

r = f:

Again we can compute SAT(M,!phi) = {S1,...,Sn} and notice that the Si will again all have the form: Si=[[ai,bi],[bi,bi]] where phi1@[ai:bi] is true in M [!phi1]@[bi+1:bi+1] is true and [!phi2]@[bi:bi] is also true. That is, bi is the point where phi1 ends and where phi2 is not true. We will refer to the bi as the critical points. Note that it is probably not necessary to run the general SAT routine to compute the critical points {b1,...,bn}. Rather given the spanning intervals for phi1 and phi2 the critical points can be easily derived from them using a specialized routine.

For each of the critical points bi we can define thre high-level moves:

1) Extending the extent of phi1 in M to satisfy the violation at bi.

To do this let t' be the first point greater than bi that ends a spanning interval where phi2 is true in M. The corresponding moves are:

Moves(M,[bi+1:t'],phi1)

Pictorially this move goes form a model that looks like:

```
      2222222
11111
```

to one that looks like:

```
      2222222
111111111111111
```

2) Shrinking the extent of phi1 in M to satify the violation at bi

Let t' be the latest point in [ai,bi] where phi2 is true or 0 if there is no such point. The moves for this option are:

Moves(M,[t':bi],[!phi1])

Note that here !phi is either a simple conjunction or disjunction of literals since phi is either a conjunction or disjunction of literals.

Pictorially this move goes form a model that looks like:

```
    2222222
11111111111111111111
```

to one that looks like:

```
    2222222
111111111111
```

3) Asserting that phi2 is true for some spanning interval ending at bi

To do this let t' be the latest point in [ai,bi] where phi2 is true in M if any or be bi if there is no such point. The corresponding moves are:

Moves(M,[t'+1:bi],phi2) or if t'=bi just Moves(M,[bi:bi],phi2)

Since phi2 is assumed to fall in our syntactically restricted space we can use the above procedure to compute the moves.

Pictorially this move goes form a model that looks like:

```
     2222222
1111111111111111111
```

to one that looks like:

```
     222222222222222
1111111111111111111
```

---

*Formula Type 3: (phi1 ; ... ; phik) -> \D_{mi} (phik | phi'), where phi' is a simple liquid formula and the phi1,..,phik are simple conjunctions or disjunctions.

Given the above structure we can compute SAT(M,!phi) = {S1,...,Sn}. The Si in this case will not be liquid, but they will all have the same form, in particular, Si=[[ai,bi],[bi,bi]] where phik ends at bi. Again, it is likely better to develop a specialized routine to compute the critical points {b1,...,bn}.

For each of the critical points bi we can define three high-level moves, which resemble the moves for Type 2 formula with r=mi:

1) Extending the extent of phik in M to satisfy the violation at bi.

To do this let t' be the first point greater than bi where phi2 is true in M. If there is not such point then (1) generates no moves.The corresponding moves are:

Moves(M,[bi+1:t'-1],phik)

Since phi1 is a simple liquid formula we can use the "moves" procedure described above to compute this set of moves.

Pictorially this move goes form a model that looks like:

```
111122223333...kkkk
              pppp    ; 'p' indicates where phi' is true
```

to one that looks like:

```
111122223333...kkkkkkkkk
              pppp
```

2) Shrinking the extent of phik in M to satify the violation at bi

Let t' be the latest point in [ai,bi] where phi2 is true or 0 if there is no such point. The moves for this option are:

Moves(M,[t'-1:bi],[!phik])

Note that here !phi is either a simple conjunction or disjunction of literals since phi is either a conjunction or disjunction of literals.

Pictorially this move goes form a model that looks like:

```
111122223333...kkkkkkkkk
              pppp
```

to one that looks like:

```
111122223333...kkkkkk
              pppp
```

3) Asserting that phi' is true for some spanning interval starting at bi + 1

To do this let t' be the first point greater than bi where phi2 is true in M if any or be bi+1 if there is no such point. The corresponding moves are:

Moves(M,[bi+1:t'-1],phi')

Since phi2 is assumed to fall in our syntactically restricted space we can use the above procedure to compute the moves.

Pictorially this move goes form a model that looks like:

```
                  pppppp
111112222...kkkkkk
```

to one that looks like:

```
              pppppppppppp
111112222...kkkkkk
```

So the full set of moves is the union across all bi of the three sets of moves defined for each bi. Again here there is the choice of evaluating them all and picking the greedy one, or just a subset of them.

---

*Formula Type 4: phi' -> !(phi1 ; phi2 ; ... ; phik), where phi' and phii are conjunctions or disjunctions of literals

If this formula is violated then there must be some intervals where phi' is true and also phi1;...;phik is true. These spanning interval will have the form Si=[[ai,bi],[ci,di]], where phi1 is true during [ai:bi] and phik is true during [ci:di].

For each Si let Si1,...Sik be the spanning interval for which phii is true over during the occurence of phi1;...;phik in Si.

There are two main types of moves for any Si:

1) Break the occurence of phi1;...;phik

The corresponding moves are:

Union_{j=1,..,k} Moves(M,Sij,!phij)

Since phij is a simple conjunction or disjunction these moves are simple.

Pictorially these moves look like:

```
111112222233333
ppppppppppppppp
```

moving to

```
    2222233333
ppppppppppppppp
```

or

```
11111   33333
ppppppppppppppp
```

or

```
1111122222
ppppppppppppppp
```

2) Reduce extent of phi'

The corresponding moves are:

Moves(M,[ai:bi],!phi') + Moves(M,[ci:di],!phi')

Pictorially these moves look like:

```
111112222233333
ppppppppppppppp
```

moving to

```
111112222233333
```

pppppppppp

or

111112222233333
pppppppppp

---

*Formula Type 5: phi1 -> \D_{mi} (phi1 | [phi2 ; phi3 ; ... ; phik]), where the phii are conjunctions or disjunctions of literals

This is the most complex of the formula types we've seen wrt defining moves. This is because there is significant ambiguity about the transition points of phi2;...;phik.

Each violation of this formula will be of the form of type 2, i.e. Si = [[ai,bi],[bi,bi]]. Each critical point bi occurs at the end of a phi1 interval that is not met by the phi2;...;phik sequence.

Two simple case is when either phi2 ; ... ; phik occurs after bi at a point t. In this case we can expand phi1 using

1) Move(M,[bi,t],phi1)

similarly if phi2; ... ; phik occurs in [ai:bi] starting at t then we can shrink phi1 using

2) Move(M,[t,bi],!phi1)

When phi2;...;phik does not occur as above the situation becomes more complicated. The trivial solution is to remove phi1 completely:

3) Move(M,[ai:bi],!phi1)

but otherwise we need to insert a phi2;...;phik sequence that starts at bi+1. The number of ways this can be done is exponential in k and we can't consider them all explicitly. One approach would be to define an objective function and define a Viterbi style algorithm for computing a transition sequence that maximizes the objective. For example, the objective would likely correspond to how well the transition sequence matched the current model (minimum edit distance). The key would be to try and optimize this in a way that does not scale with the duration of the model, which I think can be done, though not trivial. I don't think we should implement something like this yet since it is rather expensive to implement and relatively computationally intensive for a move generator.
Rather I suggest that we implement a simpler heuristic approach, get some experience with it, and expand on it if necessary. I have a simple forward expansion algorithm in mind that starts at bi+1 and then moves along end-points of spanning intervals, applying moves to make the phii true in sequence. We can go over this sometime.