Time-ordered Forward Deduction

A Bottom-up Evaluation Strategy for Starlog

Zhonge Xiao
John Cleary
Brian Unger

January 12, 1995

Contents

Abstract

Starlog, a Horn clause logic programming language, allows for time stamps, interval arithmetic, and negations. A bottom-up evaluation strategy, called "Time-ordered Forward Deduction", for the language is described. The algorithm heavily relies on time stratification as its basic execution control mechanism. A conservative approach has been adopted to handle negations, which defers the evaluation of negations until it is safe to do so. Various techniques to deal with subtle cases of inter-clause relations in negation evaluation are discussed. To illustrate the algorithm, some simple but non-trivial example programs are presented.

1 Introduction

This new evaluation strategy for logic programming language Starlog is based on previous work of Starlog, a temporal logic programming language. It has been motivated by the desire to find an approach which could be implemented efficiently enough to be usable in the future as a basis for a practical temporal logic language, most probably some kind of special purpose logic language underlying some application systems, and which would also be able to develop the full potential of Starlog as a temporal logic language, its elegance in expressing persistence by interval time values and negation. The basic algorithm presented here was first outlined out of following main considerations about Starlog execution mechanism and has been refined in the process of implementation and revised several times in the effort to solve subtle situations which were encountered when trying to run sample Starlog programs of different types, especially when dealing with negations.

- Starlog is a logic programming language which includes explicit time representation and interval arithmetic. This solves one of the major problems of logic programming – the expression of mutation and persistence. Time ordering is essential and should play a dominant role in the execution mechanism, otherwise it would be very difficult to force time order because of the very procedural nature of the time-ordering.

- The computational model of Starlog is quite different from Prolog. While Prolog is query-driven and its evaluation strategy is backward inference starting with the goals to be resolved, the answers of a Starlog program is the set of all derivable tuples. Each tuple has a time stamp which specifies the time instant or time interval over which it is true. Starting with initially existing tuples, or facts in terms of real world, the system works forward through time to generate tuples with increasing time stamp. It is natural and effective to evaluate Starlog programs bottom up. That is using existing tuples as inputs to all clauses that have in their body at least one atom of the same predicate name as that of the input tuples to derive new clause instances and eventually new tuples. This process will be repeated until all the tuples have been used and no more new tuples could be derived. The causality principle, which must be observed by all user-defined Starlog clauses, lends itself naturally and effectively to the evaluation of Starlog programs in a bottom-up fashion.

- Negation is an essential part of Starlog for such things as expressing persistence. However, with bottom-up we are basically in such a situation as waiting for something which never comes up if the negation eventually turns out to be true or which does not come up until the negation becomes false. There must be some control or ordering to facilitate solving negations, otherwise it is very easy to get lost in finding the truth or falsity of negations. Ordering by time stamps will greatly ease the task of dealing with negations.

The different new approaches of evaluation strategy that we have developed for Starlog all share the same features:

- forward deduction has been adopted as basic strategy and new clause instances will be derived only when there is a tuple that could be used as an input to the clause body so that dangerous situation of exponential explosion of the number of clause instances can be avoided. That is to eliminate unnecessary intermediate clause instances between original clause and its derivable tuples, or in other words the instances before the clause is reduced to tuples. This is essential to an efficient implementation of Starlog system.

- When a tuple is used as input to all clauses concerned to derive new clause instances, we say the tuple is being executed or "*fired*". The execution of tuples must be scheduled in the order of their time stamp. This is so important that otherwise the semantics of temporal logic would not be observed and the correct evaluation of negations extremely difficult if not impossible.

2 Basic Algorithm

2.1 Terminology

It will be helpful to introduce some terms before giving the algorithm.

tuple a unit clause (clause with an empty body) or a clause with its body containing only primitive atoms (system-defined predicates). The latter is also called constrained tuple and usually the body of a constrained tuple will be evaluated before it is issued as a tuple. A tuple represents a temporal fact which is true at certain time instance or over some period of time. Tuples are not necessarily grounded, especially constrained tuples are often not grounded as they represent equations of arguments.

pending tuple a tuple with negations pending to be evaluated. The conditions on which the tuple is true are specified by negation and subject to the result of the evaluation of negation.

time stamp the first argument of a user-defined clause is designated as time stamp of the clause. It can be a real number or an interval of real numbers. The former is said to have a "*point value*" specifying a fact that is true at certain instance of time. The latter specifies a fact that is true over a period of time. For interval time stamp its lower and upper limits are called lower boundary and upper boundary respectively and both could be open (excluded) or closed (included). tuples are always ordered by lower boundaries of time stamps if they have interval time stamp.

current time the value of time stamp of a tuple being executed or fired. If the time stamp is of interval value then current time is set to its lower boundary.

next time the value of time stamp of the tuple that has the lowest time among all existing tuples that remain to be executed.

2.2 Causality Principle

The algorithm is based on the following assumption which is essential to the correctness of the algorithm.

> **The causality principle must be observed by all Starlog programs:**
>
> > *the time in the head of a clause must be later than or at least same as the time of any atoms in the body of a clause in both its positive and negative part (clause body and negation body respectively).*
>
> This is equivalent to that the consequence of any tuple (or event in real world) must lie in its future.
>
> **Another condition concerning negation is:**
>
> > *in situations where cyclic dependency of inter-clause relations exists, any negation must be accompanied by a finite time advance.*

To be more precise, suppose `Th` be the head time stamp for a clause with point time value, `Thl` and `Thu` be lower and upper boundaries of head time stamp for a clause with interval time value; `Ti` be the time stamp or lower time boundary (if of interval value) of any user-defined atoms in a clause body including negation, the following must be held true for any clauses:

```
Th >= Ti   or   Thl >= Ti                        (1)
```

It seems that at first glance there will be some difficulty in expressing events in the future as it is often both necessary and convenient to express such things as "*the fact is true until some time in the future*". For example a piece of data in a database system keeps the same value from one update until next update. With negations those situations can be specified in the opposite way and condition (1) can be observed quite comfortably. For example, *data "key"* holds *"value"* until next *update* at time `T0` could be expressed in Starlog like:

```
data(T key value) <- not( update(T0 key newvalue), T>T0 ).
```

2.3 General Description

Two working list structures (tuple list and pending tuple list) play central roles in Starlog execution scheduling, therefore they are important to the understanding of the algorithm.

tuple list a list that keeps all the tuples, initially existing or generated during the execution of a Starlog program, in non-decreasing time order. Each member of the list is a pair of tuple and its time tag. The time tag of a tuple equals to its time stamp value. The list is ordered by lower boundary of time stamp. The tuple list grows as new tuples with later times are generated, and obsolete tuples with earlier time are collected away from time to time.

pending tuple list a list that keeps all the pending tuples in non-decreasing time order (same as tuple list). Each member of the list consists of the tuple, its negative part, its time stamp, and some other information for optimized evaluation of negation. Newly generated pending tuples are added to the list and resolved tuples are taken away from the head of the list.

A general description of Time-ordered Forward Deduction algorithm follows:

Initial tuples given by a user program are collected and put onto tuple list at the initialization phase;

Execution starts with the tuple with lowest time stamp (the first member of tuple list) and proceeds one after another towards the end of the tuple list;

Each step of execution involves only one tuple, called current tuple or input tuple. It is used as input to all clauses which contain in their body at least one atom with the same predicate name as the current tuple's. The process of using a tuple as input to reduce a applicable clause instance one step closer to its final result – tuple is called "*derivation*". At each execution step, some new tuples may be generated and issued as part of the program solutions, or new clause instances may be derived and stored for further derivation;

The derivation process proceeds as follows:

> For each distinct occurrence in a clause body of the atom with same predicate name as the input tuple's, a unification is applied to the atom and the input tuple. If unification succeeds, the new variable bindings are propagated and the atom deleted (resolved), or replaced by the input tuple's body if the tuple body is not empty. If the resulting new clause instance is a unit clause or its body contains no more user-defined atoms, than a new tuple has been generated and is added to the tuple list in time order (after existing tuples with same lower boundary of time stamp), otherwise it is just a new clause instance of one step closer to its derivable tuples.
> This process is repeated for each occurrence of the predicate in all clauses until all the possible tuples are generated or new clause instances are derived. Notice that two occurrences in the same clause with different arguments are considered two distinct occurrences and will result in two different clause instances;

Execution terminates when it reaches the end of the tuple list and no more tuples are generated.

Current time is updated each time a tuple is taken as input tuple from the tuple list. There is a clear cut between the current time and earlier times. It is guaranteed that all the tuples of earlier time than current time have been generated, in other words, no more new tuples of earlier time will ever be generated. This is obvious according to condition (1) and from the execution scheduling strategy described above.

Suppose `Tc` is current time, `Tn` is next time (if current tuple is the last one in tuple list then next time is set to infinite), `Tin` is the lower time boundary of any input tuples from now on and `Tpi` is the lower time boundary of any tuples which will be generated as consequences of input tuples,

```
since:  Tn >= Tc and Tin >= Tc
and from condition (1):
        Tpi >= Tin                              (2)

so      Tpi >= Tc                               (3)
```

This is extremely important to the negation evaluation strategy described later for it is safe to decide based on the current time whether a negation is true or false up to now.

2.4 Dealing with Negations

A unique feature of the inference strategy for Starlog presented here is the way it handles negations. Negations are not evaluated until the time comes when negations can be resolved definitely, that is when enough evidence (facts or tuples in Starlog terms) has been collected. This will greatly simply the handling of negations and is a major contributing factor to a more efficient implementation. This is also enforced by the time ordering requirement of the algorithm presented above. Negation evaluation relies on all possible tuples and if the result of evaluation is true, the resulting tuple, which could be drawn from the pending tuple, must also comply with the time ordering requirement.

The general algorithm for negation handling is given as follows.

In the derivation process described above, the negations, if any, in the clause being considered are totally ignored. In other word, the execution of a clause body is separated from the evaluation of its negation part. However, it does matter when the clause becomes a "would-be" tuple (it would be a tuple without negation) called pending tuple. The truth or falsity of a pending tuple is conditional depending on the truth or falsity of the negation.

Pending tuples are put onto the pending list in non-decreasing time order. Before the current time advances to next time, if `Tn > Tc`, the system checks against `Tpl`, the lower time boundary of the first member of the pending list, if the following condition met, then the execution of tuple is suspended and the evaluation of negation is initiated for the first member of the pending list.

```
Tpl >= Tc and Tpl < Tn                          (4)
```

The result of evaluation is either true or false. If true then the tuple does exist and is added to the tuple list, if false, then it is not a true tuple and simply deleted from the pending list.

The negation evaluation process will be carried out for as many pending tuples that satisfy condition (4) as there are in the list before the system resumes the execution of next tuple. Notice that new generated tuple as result of negation evaluation may well have earlier time than `Tn` and thus it will change the value of `Tn` for next round of evaluation.

It is not difficult to draw some important conclusions from this negation evaluation algorithm.

A. From condition (4), any pending tuples with time earlier than `Tn` will have been resolved before `Tc` advances to `Tn`, so at any time any pending tuples in the pending list may only have their time stamp:

```
Tpl >= Tc                                        (5)
```

As a result we have:

```
Tpi >= Tc                                      (6)
```

where `Tpi` is the lower time boundary of any tuples which can be drawn from pending tuples. So with this negation algorithm time ordering requirement is also satisfied.

B. From conclusions (3) and A above, it is assured that no tuples will ever be generated with time `Tpi < Tc`. It can be concluded:

> *For pending tuples with point time value and all the user-defined atoms in their negation bodies having time stamp `Tai and Tai < Tc`, it is safe to say that negation is evaluated in aware of all relevant tuples and therefore the results of negation evaluation are correct.*

For more general cases such as interval time stamp and `Tai=Tc`, conclusion B. may not be true and the above general algorithm may not be able to cope with these subtle, complex situations. More elaborate mechanism as will be discussed in later sections of this report is needed.

2.5 Example Program 1 – prime number generator

To demonstrate the basic algorithm presented in this section, a program which generates all the prime numbers in increasing order is given along with the tracing of program execution for several tuples. This program has only two clauses and one initial tuple which says "*2 is a integer*". Clause *int* is actually an integer generator starting from 2. Clause *prime* is defined elegantly by negation which says "*if not exist such two integers `I` and `J` that their multiplication equals to integer `N`, then `N` is a prime number*". This is actually the definition of what is a prime number.

Here is the program:

```
int(2).
```

```
int(N) <- int(M), N=M+1.

prime(N) <- int(N),
            not-exist(int(I), int(J), N=I*J).
```

In this program integers are taken as a kind of logical times. Arrow `=>` points to current input tuple being executed or the pending tuple under evaluation. The execution is scheduled in strict time order.

```
==========================================================================
tuple list      derived                             pending list
--------------------------------------------------------------------------
  Tc= 2
=>int(2)      int(3)<- int(2),3=2+1.                empty
              prime(2)<- int(2),
                         not-exist(int(I),int(J),
                                   2=I*J).
--------------------------------------------------------------------------
  A new tuple int(3) and a pending tuple prime(2) are derived by firing
  input tuple int(2).

==========================================================================
tuple list      derived                             pending list
--------------------------------------------------------------------------
  Tc= 2, Tn= 3                                    =>prime(2)<- not-exist(
  int(2)      negation succeeded                                int(I),
  int(3)      (no integers less than 2)                         int(J),
              tuple drawn:                                      2=I*J).
                prime(2)<-
--------------------------------------------------------------------------
  Evaluating negation of due pending tuple before advancing current time
  Tc to next time Tn,
  negation succeeded because no integers less than 2 exist so negation
  body could not be satisfied, tuple prime(2) has been drawn from the
  pending one.

==========================================================================
tuple list      derived                             pending list
--------------------------------------------------------------------------
  Tc= 2
  int(2)        no applicable clauses.              empty
=>prime(2)
  int(3)
--------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------
  Tc= 3
  int(2)      int(4)<- int(3),4=3+1.
  prime(2)    prime(3)<- int(3),
=>int(3)                 not-exist(int(I),int(J),
                                   3=I*J).
--------------------------------------------------------------------------
  input tuple int(3) has resulted in tuple int(4) and pending prime(3).
==========================================================================
tuple list      derived                           pending list
--------------------------------------------------------------------------
  Tc= 3, Tn= 4                                  =>prime(3)<- not-exist(
  int(2)      negation succeeded                                int(I),
  prime(2)    (no such I and J)                                 int(J),
  int(3)      tuple drawn:                                      3=I*J).
  int(4)        prime(3)<-
--------------------------------------------------------------------------
  Negation succeeded again because no integers could satisfy 3=I*J,
  another tuple prime(3) has been drawn from the pending tuple.
==========================================================================
tuple list      derived                            pending list
--------------------------------------------------------------------------
  Tc= 3
  int(2)      no applicable clauses.               empty
  prime(2)
  int(3)
=>prime(3)
  int(4)
--------------------------------------------------------------------------
  Tc= 4
  int(2)      int(5)<- int(4),5=4+1.
  prime(2)    prime(4)<- int(4),
  int(3)                 not-exist(int(I),int(J),
  prime(3)                         4=I*J).
=>int(4)
--------------------------------------------------------------------------
  Tc= 4, Tn= 5                                  =>prime(4)<- not-exist(
  int(2)      negation failed                                   int(2),
  prime(2)    (2*2=4, 4 is not a prime number)                  int(2),
  int(3)                                                        4=2*2).
  prime(3)
  int(4)
  int(5)
--------------------------------------------------------------------------
```

This time, negation failed because 4=I*J can be satisfied with I=2 and J=2, so 4 is not a prime number, no tuple could possibly be drawn.

```
==========================================================================
tuple list      derived                           pending list
--------------------------------------------------------------------------
  Tc= 5
  int(2)      int(6)<- int(5),6=5+1.              empty
  prime(2)    prime(5)<- int(5),
  int(3)                 not-exist(int(I),int(J),
  prime(3)                         5=I*J).
  int(4)
=>int(5)
--------------------------------------------------------------------------
  ...       ...                                   ...
```

For this program, the time stamps I and J in negation body could only have values less than the time stamp N in the head. Since all the int tuples with time value less than N have been generated when evaluating negations, the correctness of negation evaluation is guaranteed.

3 Stratification on Time Intervals

3.1 Identify Cases of Stratification

The algorithm described above organizes the execution of program in strict time order, we can say that the algorithm itself heavily relies on on stratification on time, but here we are talking about stratification on time intervals.

There are two situations which give rise to the need for stratification on time intervals. These all happen to pending tuples with time stamps of interval value. Pending tuples are derived by executing clauses with negation. Upon the generation of a pending tuple with interval time, its lower boundary of time stamp is usually fixed as the result of execution, while its upper boundary is in most cases unknown or "open" and subject to the conditions specified in its negation part. This is typical of the database case where a piece of data once updated will keep its value until next update in the future. That is the upper boundary of time stamp or exact duration over which the fact that a pending tuple expresses is true depends on some events in the future, or in Starlog term some tuples which will be, but not yet generated. This is contradictory to the negation strategy described in previous section. The problem lies in that the time ordering is in any circumstance always by lower boundary of time intervals so that the strict time order of tuple execution can be observed. On the other hand, the upper boundaries vary and stretch to the future times. The Starlog interpreter can only evaluate negations base on the conditions up until now as is enforced by the time ordering requirement of the basic algorithm for Starlog inference mechanism.

Now we discuss the two cases and the way to deal with them respectively. The first situation arises when the result of negation evaluation for a pending tuple is true, or negation succeeds (negation body could not be satisfied). Since this conclusion is merely based on conditions up till now, it may only be said that "*until now it is true, but don't know whether it will continue to be true*". In this case we need to stratify the time interval and leave the "future" to be decided later.

The second situation arises when the result of negation evaluation fails or succeeds partially (not over the entire time interval). It is necessary then to break down the whole time interval into smaller pieces, to leave the future piece, if any, in the pending list, and a tuple that is true over portion of the time interval generated.

3.2 Success of Negation

In first situation, the rules for stratification are as follows.

> **The time value of a negation being evaluated is a interval and the result of evaluation is true, then further identify three cases:**
>
> 1. if the lower boundary (closed) equals to the current time, then stratify the time interval into two pieces, one has the current time as its time stamp (of point value) and the other one has the same time interval value but with its lower boundary open.
> 2. if the lower boundary (open) equals to current time, or the lower boundary (open or closed) is greater than current time, and the upper boundary is greater than or equal to next time, then stratify the time interval into two pieces, making next time as first one's upper boundary (open), the rest for the second piece.
> 3. if the lower boundary (open) equals to current time, or the lower boundary (open or closed) is greater than current time, and the upper boundary is less than next time, then no stratification is imposed.

It may not be necessary to identify first case from second one, but it is included in consideration of implementation optimization for it is often the situation that the current time is critical to the negation under evaluation. In the second case, the next time Tn may be irrelevant to the pending tuple at all, but it is unknown until current time that what tuples might be generated starting from Tn. This could cause some efficiency problems in certain applications.

3.3 Partial Failure of Negation

In second situation of stratification, negation is not failed or succeeded over the entire time interval of a pending tuple, but on portion of it. Before giving the general rule for stratification on time interval in this situation, it is worthwhile to spare some space discussing an important property of the interval arithmetic built into Starlog.

The semantics involved in variable bindings in Starlog is standard as that of satisfying Prolog goals:

- Instantiated variables serve as input parameters in satisfying goals, they can not be re-instantiated and serve as output on return of a call to a goal.
- Uninstantiated variables will probably be instantiated acting like output parameters;

There is an exception with Starlog, however, those variables that are instantiated to arithmetic interval values may well be further instantiated ("squeezed") in unification or primitive calls. These variables serve as both input and output parameters. This bears an important implication when it comes to solve negations. The evaluation (or execution) of a negation is in fact to try to satisfy the goals in negation body. Any modification, or "squeezing" made to time intervals after the evaluation simply means that negation body is "**partially**" satisfied, so negation is failed not on the entire time interval but only on portion of it.

In general the result time interval of partial negation failure would be like what the following simple figure shows,

where `Tpl` and Tpu are lower and upper boundaries of original time interval of the pending tuple, the negation failed only on the portion of original interval that has `Tpl'` and `Tpu'` as its lower and upper boundaries respectively. Stratification is then carried out by breaking down the original time interval into three pieces of time fragment:

1. first piece is calculated from the difference between `Tpl and Tpl'`, and it could be of a interval value, or of a point value, over which negation

```
-------------------
Tpl   Tpl'   Tpu'  Tpu
```

Figure 1: Partial Failure of negation

succeeded and a tuple can be generated with this time fragment as its time stamp. This piece could also be "**nil**", no tuple would be generated then;

2. since negation failed over second piece of time fragment, no tuple could be generated over this time period;

3. the third piece has a "future" time, over which the negation may not be true should some tuples generated later against it, as such it should remain in the list in place of the original pending tuple awaiting for further investigation.

There are some cases, however, which also leaves the failure over some time segment in question. Here is a example, a clause defines a predicate P true over some time interval `Tp` if `P` is not true over time interval `Ts` such that `Tp = Ts + 1`.

```
P(Tp) <= not(P(Ts) Tp=Ts+1).
```

In other words, if `P` is true over Ts then it can not be true over a time interval which can be obtained by shifting Tp 1 unit further. Given `Tp>=0`, the initial tuple would be `P(Tp:[0 w>)`, that is `P` is true over time from 0 to infinite. With this tuple, the negation would fail over `[1 w>`. So tuple `P(Tp:[0 1>) (Tp>=0 and Tp<1)` is realy true, but for time interval `[1 w>`, `P` may not fail entirely as when `P([0 1>)` is fed back to negation, the result would be that `P` is not true over `[1 2>` and may true over `[2 w>`. This process would repeat for ever and yield tuples `P([0 1>)`, `P([2 3>)`, `P([4 5>)`, ... The problem here is caused by recursive "*referencing*" in negation body. To resolve this problem, a simple method is adopted. When `Tpu'=Tpu`, a pending tuple with time value ranging from `Tpl'` to `Tpu'` will remain in the pending list waiting for future evaluation, although it has failed based on currently available facts.

3.4 The Validity

In case that the negation failed entirely, the pending tuple is deleted, no stratification, nor any tuple could be generated at all, because there is already enough evidence that negation is false. This, however, may only be concluded under the assumption that once a tuple is generated, it may never change as opposed to an optimistic evaluation strategy which is presented in the third report on *Time-ordered Forward Deduction* algorithm for Starlog.

The main point here is that for negation NEVER "*look ahead*" because it is unknown for the "*future*", instead always see what already exist and make sure every possible situation has been considered so that it is safe to make a decision on whether the negation is true or false. This is only justified if the following assumption is held true:

> *All the tuples drawn from the results of negation evaluation before next time Tn do not in turn affect the truth or falsity of negation, in other words they could not have contributed to the negation evaluation.*

With and only with this assumption the stratification rules given above can be carried out safely, otherwise it could be erroneous. Unfortunately, probably in most cases this assumption does not apply because recursive definitions are extensively used in programs written in logic language. This will be the topic of section 4.

To be more confident, some informal reasoning may help.
By the time to evaluate negations for pending tuples, all the tuples that have time (lower boundary) less than Tn and which could possibly be derived by firing input tuples from the tuple list are already generated sitting in the tuple list. Suppose Tpi represents the time stamp of any tuples that will be generated in the future by executing input tuples, then we have

```
Tpi >= Tn                                   (7)
```

So we may conclude that with the above assumption, the negation evaluation and the stratification rules given here are correct.

4 Critical Phase

4.1 The Problem

The assumption made in last section has to be lifted for the system to be able to handle any serious negation cases. This, however, will challenge the philosophy of negation evaluation – *"things are only clear up till now, but not sure at all about future"*. The situation which arises with stratification discussed in previous section virtually requires looking into future in some way in order to stratify time intervals correctly. Let's look at an example program called "colliding pucks" which models the movement of pucks on a billiards board. To solve this seemingly simple problem in more conventional language requires a surprisingly complex algorithm. With Starlog it is easy by negation calls to say things like that a fact will be true for a period of time until some event occurs, so the Starlog program for this problem becomes quite simple. The main part of the program consists of three clauses, one specifies the trajectory of a puck, the others specify when collision occurs. Here are the simplified version,

```
trajectory(T,N,V,X) <-  collide(T0,N,P,V), T>=T0,
                        X = P + (T-To)*V,
                        not-exist(collide(T1,N,P',V'),
  T1>T0, T>T1).

collide(T,N1,X,V1new),collide(T,N2,X,V2new) <-
                        trajectory(T,N1,V1,X),
                        trajectory(T,N2,V2,X),
                        N1 < N2,
```

```
                                physics(V1,V2,V1new,V2new).

    collide(T,N,X,Vnew) <-  trajectory(T,N,V,X),
                            wall(X,M),
                            Vnew = V*M.
```

We are not interested in exact meanings of these clauses, it is suffice to say that clause *trajectory* expresses that if puck `N` at time `T0` and position `P` had a collision, and the after collision velocity is `V`, then puck's position at any time `T` after `T0` can be calculated by

```
X = P + (T-T0)*V.
```

And this trajectory remains valid until the puck collides again at time `T1`. Given trajectories, the first *collide* clause specifies when two pucks collides. Goal *physics* is to calculate after collision velocities by physics laws. The second *collide* clause specifies when a puck collides with wall.

While the declarative nature of Starlog language makes this program simple and easy to understand, it presents no easy task for the interpreter to handle the negation behind the scenes. Since the collisions are to be determined by trajectories, no collision tuples which would be generated in the future could be seen when evaluating the negation. With the negation strategy given in previous sections, a *trajectory* tuple with upper time boundary open (set to infinite possibly) would be generated, this in some sense is correct because it gives the trajectory without encountering any collision. With this trajectory the next collision could then be calculated which in turn would change the puck's trajectory. More than one collision might be calculated with the same puck trajectory, but only the earliest collision could actually happen because the trajectory would then change to a new one, so the upper time boundary of the trajectory tuple must be modified and other possible collisions (probably these "collide" tuples were already in the tuple list) along the trajectory had to be reconsidered.

The problem posed by "colliding pucks" program is typical and it requires looking into the future, this is a very complex and subtle situation which can not be dealt with efficiently without some knowledge of the structure of the actual program under execution. Let's look at the program again and pay attention to the structure pattern. The *trajectory* clause contains a negation with a atom *collide*. The *collide* clauses in turn contains *trajectory* atom or atoms in their body. This is to say the *trajectory* extends until next "collision" and the next "collision" will in turn terminate the old "trajectory" and set off a new "trajectory". This is a common scenario in real world. The point here is that the pattern of interrelations between clauses forms a closed path or circle path. As such, the pending tuples of *trajectory* will generate *trajectory* tuples with unknown end points, these 'trajectories" will result in "collision" and generate *collide* tuples and the latter will in turn restrain the *trajectory* tuple. In terms of time ordering, the point is that it is possible that some tuples with time earlier than next time Tn may be drawn from pending tuples and these tuples will affect the duration of the truth of the original pending tuples. This is caused by recursive definitions of predicates which is common in logic programming.

One solution to this problem is to find out all the tuples with time stamps earlier than Tn, which could be drawn from due pending tuples, and to reevaluate negations based on these new information about possible tuples in a bid to get correct stratification on time intervals or simply to draw correctly true tuples from pending tuples. The notion of "**Critical Phase**" has been developed out of this idea.

Before presenting the mechanism for handling Critical Phase execution, we will discuss paths analysis first.

4.2 Paths Analysis

One of the key issues involved in "**Critical Phase**" mechanism is the analysis of some aspect of program logic structure, i.e. to know something specific about the program under execution. As mentioned above, we are particularly interested in inter-clause relations which could give some clue on whether or not the correctness of negation evaluation would be affected by the result of a pending tuple itself or some other due pending tuples.

The inter-clause relations are represented by directed graph. A node in the graph represents a clause. If the clause is negated (a clause with negation),

Figure 2: Directed Graph representing inter-clause relations

part of the node is blackened. Directed arcs represent input of tuple. An arc drawn from node A to node B, as the following figure shows, means that the tuple derived from clause A can be an input tuple to clause B, in other words clause B contains at least one atom in its body that has the same predicate as that of clause A defines. If the coming arc points to blackened part of a node, then the input is to the negation body instead of the positive part of the clause.

Four basic patterns of inter-clause relations, which all start from a negated clause, can be recognized as given by figure 2. Type (1) forms a **closed path** or **circle path** which reflects recursive definition of predicates. Part of type (2) forms closed path, but it is irrelevant to the negation evaluation. In type (3) the path ends with a node whose tuples can not be a input to any clauses and the path comes to the positive body of the last node. Type (4) is almost the same as type (3) except that the path comes to the negation part of the clause of last node. The real inter-clause relations can be much more complicated, it may combine different types. Among these four types only type (1) and type (4) are relevant to negation evaluation and they possess the same properties. They all start with a negated clause and come at the end to the negation body of the same or another negated clause. Type (4) is called "**cross path**" meaning one negated clause is related to another negated clause and its result may effect another especially when they happen to have same lower boundary of interval time.

A pending tuple derived from a clause involved in these types of interrelation could generate a tuple whose execution could in turn affect the result of negation evaluation of itself or affect another pending tuple's result.

Further more we can group together all the paths that end with the same negated clause and we call this group of paths "**Related Paths**" to that **negated clause**. It is easy to see that a negated clause may have no **Related-paths** in which case its negation evaluation is immune from the

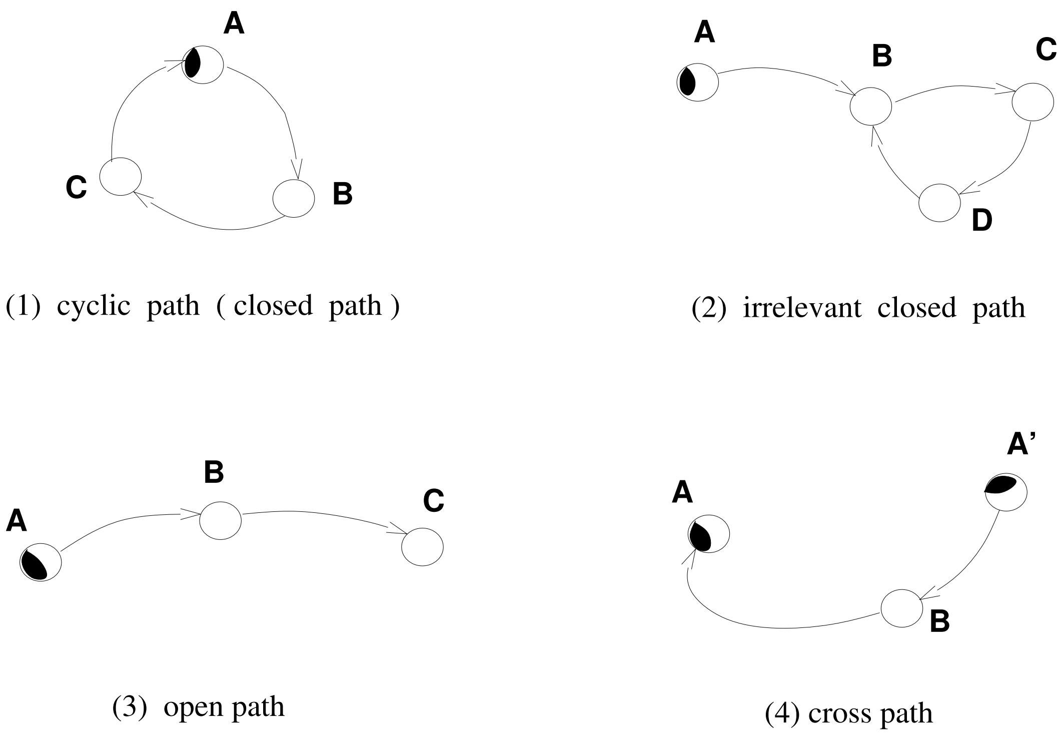

Figure 3: Basic Types of inter-clause relations

interference of its own result or any other pending tuples', or a negated clause may have **Related-paths** that contains one or more paths, either **circle path** or **cross path** or both.

The analysis of inter-clause relations graph of a program can be carried out at initialization stage prio to the execution of the program. The purpose of this analysis is to find out Related-paths for each negated clause. The result information should be collected and saved for use in the Critical Phase execution.

4.3 Critical Phase Execution

The notion of "**Critical Phase**" and a mechanism for dealing with it is the topic of this sub-section.

In general, the evaluation of negations is not carried out until the time when enough evidence (facts or tuples in Starlog terms) has been collected. When all the tuples with time earlier or equal to `Tc` have been fired (used as input to relevant clauses to derive new clause instances, or tuples if all the user-defined atoms in a clause body have been unified with some input tuples), now it is the time to evaluate negations if there are any tuples pending (tuples with negation unsolved) and their time `Tpl` satisfies the following relations:

```
Tpl >= Tc and Tpl < Tn.                           (8)
```

There might be more than one pending tuple that meet time condition (8), they are all **due pending tuples**. Depending on certain conditions, the system may have to enter **Critical Phase** where subtle situation could arise making the evaluation of negation complicated. These conditions are:

- the result of negation evaluation of a due pending tuple is true;
- a pattern of inter-clause relations has been detected by analyzing the graph of inter-clause relations, which bears the potential that some tuples drawn from the pending tuples under evaluation and having time stamp less than Tn may well in turn affect the correctness of original evaluation of negation.

When the above conditions are met, then we say that "*the system has entered **Critical Phase***" meaning the executions during this phase are tentative and some results of executions might not be true.

The first condition actually says that from what are known until now the negation is true. It is not necessary to enforce Critical Phase in the case of negation failure. If the negation evaluation has failed entirely over the time interval (or time instance in the case of point value), then it is clear that enough "**evidence**" or tuples of earlier time already exist that the pending tuple could not be true and there is no need to "*look into the future*". If, however, the evaluation has failed partially, then the pending tuple must have a interval time stamp. A tuple can be drawn for the first piece, if there is one, of time interval as depicted in figure 1. The third piece of time interval will remain and wait for further evaluation in the future. In this case it is unlikely that those tuples that would be generated if Critical Phase were activated could affect the evaluation result because these tuples would have later time stamps. For either of these two failure cases the result of evaluation is reliable and there is no need to resort to Critical Phase mechanism.

The second condition can be determined by examining information on Related-paths for each due pending tuple. If any Related-paths is valid, then we say that "*a pattern of inter-clause relations that may affect the negation evaluation has been detected*".

The mechanism for handling Critical Phase consists of the following steps:

- PRELIMINARY TUPLE Since the result of negation evaluation is true, a tuple can be drawn from the pending tuple under investigation (called **current pending tuple**). Stratification is carried out on the time stamp of the pending tuple as described in section 3 if the time stamp is of interval value. The tuple that can be drawn from current pending tuple is issued on the preliminary basis.

- PATHS ANALYSIS The Related-paths for the current pending tuple is retrieved and analyzed in order to find **relevant paths**. All the **closed paths** in Related-paths are relevant. A **cross path** from the Related-paths is also relevant if there is another pending tuple with time less than Tn (it is also a due pending tuple) and it has been derived from the same negated clause as the head node clause of the cross path. Once a relevant cross path has been found, a another due pending tuple will be involved and it should be evaluated immediately, and step a) and step b) would also be applied to it if the result of evaluation satisfies Critical Phase conditions given above.

  Current pending tuple along with other due pending tuples which are found relevant to the current one by cross paths, are called **relevant pending tuples**. And Only the relevant paths for each relevant pending tuple are subject to further examine in the steps to follow.

- PARTIAL EXECUTION The execution of tuples during Critical Phase is quite different from normal execution. Based on the results of previous steps, the execution in Critical Phase is selective, it is to be carried out only for relevant paths. Along each relevant path the execution proceeds as follows, use the preliminary tuple drawn from the due pending tuple, which has been itself derived from the negated clause of the head of the path, as input tuple and apply it to next node on the path, the resulting tuple of the execution will in turn be applied to one which follows on the path until it comes to the end of the path (the one next to last actually) or no tuple could be derived at any node on the path.

If there are more than one pending tuple involved, execution will be conducted for the relevant paths of each of them. This could make it easy to solve complicated inter-clause relation patterns, especially one of mutual dependent relations. All the tuples generated during Partial Execution are to be put into tuple list temporarily (or be put into a working tuple list instead).

- NEGATION REEVALUATION After every relevant path of all the relevant pending tuples have been followed by execution, if there are any tuples generated in this whole process of Partial Execution and their time stamp are earlier than Tn, then the negation part of current pending tuple should be re-evaluated with this new information about tuples and correct tuple may be drawn from the result of re-evaluation.

The **Critical Phase** ends in two possible situations.

1. No tuples that have time stamp less than Tn could be generated in Partial Execution, consequently the original negation evaluation is correct and the preliminary tuple is in deed a true tuple.
2. Tuples with earlier times than Tn have been drawn from relevant due pending tuples and negation re-evaluated. Finally correct tuple is generated in place of the preliminary tuple or no tuple could be drawn at all.

Notice that all the results of Partial Execution except the final tuple should be cleared out, no side effects should be allowed to remain as some are simply false results.

4.4 Example Program 2 and 3

The "db.sta" example has been revisited with the latest version of Starlog, and a new program "oscillator.sta" is presented to demonstrate the Critical Phase mechanism.

Note that "w" denotes INFINITE.

```
  Example 2:                       db.sta
==========================================================================
initial facts          program rules
  (tuples)               (clauses)
--------------------------------------------------------------------------
  update(1 a 4)     db(T Key V) <- update(T0 Key V), T>T0,
  input(2 incr)                    not-exist(T1>T0, T>T1,
  update(3 a 7)                              update(T1 Key X)).
                    update(2 Key W) <- db(2 Key V), W=V+1, input(2 incr).
==========================================================================

  Tuple input is not derivable because no clause defines predicate input,
  These type of tuples like input(2 incr) is executed in initialization
  phase. As a result, clause update becomes partially instantiated.
==========================================================================
tuple list          derived                         pending list
--------------------------------------------------------------------------
Tc = 1
=>update(1 a 4)     db(T:<1 w> a 4) <- not-exist(T>T1, T1>1,
  update(3 a 7)                                 update(T1 a X)).
--------------------------------------------------------------------------
  Pending tuple db(T:<1 w> a 4) is generated with upper boundary set to
  what the execution of positive part has resulted in.
```

```
==========================================================================
tuple list          derived                           pending list
--------------------------------------------------------------------------
Tc = 1, Tn = 3
  update(1 a 4)     stratify time interval into:  =>db(T:<1 w> a 4)<-
  - - - - - - -     <1 2] and <2 w>                     not-exist(
  update(3 a 7)     tuple issued:                         T>T1, T1>1,
                    db(T:<1 2] a 4) <-                    update(T1 a X)).
--------------------------------------------------------------------------

  Initial stratification was -- <1 3> and [3 w>;
  There is a circle path update->db and system entered Critical Phase;
  preliminary tuple db(T:<1 w>) issued;
  execution of clause update with input tuple db(T:<1 w>) results in tuple
  update(2 a 5);
  re-evaluation of pending tuple with tuple update(2 a 5) present;
  finally, the execution in Critical Phase has resulted in correct
  stratification -- <1 2] AND <2 w>.
  tuple db(T:<1 2] a 4) issued and tuple with time <2 w> remains pending.

==========================================================================
tuple list          derived                           pending list
--------------------------------------------------------------------------
Tc = 1
  update(1 a 4)     update(2 a 5) <-                  db(T:<2 w> a 4)<-
->db(T:<1 2] a 4)                                         not-exist(
  update(3 a 7)                                           T>T1, T1>1,
                                                          update(T1 a X)).
--------------------------------------------------------------------------

  From clause "update", tuple (update 2 a 5) is derived with input
  db(T:<1 2] a 4).

==========================================================================
tuple list          derived                           pending list
--------------------------------------------------------------------------
Tc = 2
  update(1 a 4)     db(T:<2 w> a 5)<-not-exist(     db(T:<2 w> a 4)<-
  db(T:<1 2] a 4)                    T>T1, T1>2,        not-exist(
=>update(2 a 5)                      update(T1 a X).      T>T1, T1>1,
  update(3 a 7)                                           update(T1 a X)).
--------------------------------------------------------------------------
  Another new pending tuple generated.
```

```
==========================================================================
tuple list          derived                         pending list
--------------------------------------------------------------------------
Tc = 2, Tn = 3
  update(1 a 4)     negation false on:            =>db(T:<2 w> a 4)<-
  db(T:<1 2] a 4)   db(T:<2 w> a 4) <- ...              not-exist(
  update(2 a 5)     (T1: <1 w>, (update 2 a 5)).          T>T1, T1>1,
  update(3 a 7)                                           update(T1 a X)).
                                                    db(T:<2 w> a 5)<-
                                                        not-exist(
                                                          T>T1, T1>2,
                                                          update(T1 a X)).
--------------------------------------------------------------------------

  Negation body satisfied with update(2 a 5), so pending tuple failed
  entirely.

==========================================================================
tuple list          derived                         pending list
--------------------------------------------------------------------------
Tc = 2, Tn = 3
  update(1 a 4)     stratify time interval into:  =>db(T:<2 w> a 5)<-
  db(T:<1 2] a 4)   <2 3] and <3 w>                     not-exist(
  update(2 a 5)     tuple issued:                         T>T1, T1>2,
  update(3 a 7)     db(T:<2 3] a 5) <-                    update(T1 a X)).
--------------------------------------------------------------------------

  Continue to evaluate next due pending tuple;
  initial stratification made: <2 3> and [3 w>;
  Critical Phase execution generates no tuples because execution of clause
  update failed to generated new tuple db;
  re-evaluation carried out in consideration of next tuple update(3 a 7);
  Final tuple drawn with time stamp <2 3].

==========================================================================
tuple list          derived                         pending list
--------------------------------------------------------------------------
Tc = 2
  update(1 a 4)                                     db(T:<3 w> a 5)<-
  db(T:<1 2] a 4)                                       not-exist(
  update(2 a 5)     garbage collected:                    T>T1, T1>2,
=>db(T:<2 3] a 5)   update(2 Key W) <- db(2 Key V)        update(T1 a X)).
  update(3 a 7)                        add(V 1 W).
--------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------
Tc = 3
  update(1 a 4)     db(T:<3 w> a 7)<-not-exist(     db(T:<3 w> a 5)<-
  db(T:<1 2] a 4)                     T>T1, T1>3,       not-exist(
  update(2 a 5)                       update(T1 a X).     T>T1, T1>2,
  db(T:<2 3] a 5)                                         update(T1 a X)).
=>update(3 a 7)
--------------------------------------------------------------------------
Tc = 3, Tn = w
  update(1 a 4)     negation failed:                 =>db(T:<3 w> a 5)<-
  db(T:<1 2] a 4)   (T1: <2 w>, update(3 a 7))           not-exist(
  update(2 a 5)                                            T>T1, T1>2,
  db(T:<2 3] a 5)                                          update(T1 a X)).
  update(3 a 7)                                       db(T:<3 w> a 7)<-
                                                         not-exist(
                                                           T>T1, T1>3,
                                                           update(T1 a X)).
--------------------------------------------------------------------------
Tc = 3, Tn = w
  update(1 a 4)     negation true:                   =>db(T:<3 w> a 7)<-
  db(T:<1 2] a 4)       (T1: <3 w>, no "update")         not-exist(
  update(2 a 5)     db(T:<3 w> a 7) <-                     T>T1, T1>3,
  db(T:<2 3] a 5)                                          update(T1 a X)).
  update(3 a 7)
--------------------------------------------------------------------------

  Since next time is infinite, preliminary stratification is:
        <3 w>;
  No tuples generated in Critical Phase execution and no next tuple either,
  so the initial stratification is correct one.

==========================================================================
tuple list          derived                           pending list
--------------------------------------------------------------------------
Tc = 3
  update(1 a 4)     no applicable clauses.
  db(T:<1 2] a 4)
  update(2 a 5)
  db(T:<2 3] a 5)
  update(3 a 7)
=>db(T:<3 w> a 7)
-------------------------------------------------------------------------
```

```
------------------------------------------------------------------------
Tc = w ("INFINITE"), execution terminated.
  update(1 a 4)
  db(T:<1 2] a 4)
  update(2 a 5)
  db(T:<2 3] a 5)
  update(3 a 7)
  db(T:<3 w> a 7)
=>
========================================================================
```

The following program gives a more clear picture of how the system deals with negation.

```
  Example 3:                    oscillator.sta
=========================================================================
initial facts           program rules
  (tuples)                (clauses)
-------------------------------------------------------------------------
                    p(T) <- not(p(S) T=S+1).

=========================================================================
```

The only clause this program has, actually the whole program is, just a pending tuple. Initially the T is set to [0 w> because time stamps assume values ranging from 0 to infinite.

```
STEP 1
=========================================================================
tuple list              derived                 pending list
-------------------------------------------------------------------------
current time: 0
=>p(T:[0 1>)            nil                     p(T:[1 w>)<-not(p(S:[0 w>)
                                                                T=S+1).
-------------------------------------------------------------------------
```

Whenever a pending tuple is to be put into the pending list, a procedure is invoked to figure out by executing its primitives in negation body the possible value range of time stamps in its negation part. This is necessary both for optimization and for garbage collection.
For this particular program, the pending tuple is initial, not derived and it is, as a rule, treated in initialization phase. And it has been found immediately that p must be true over time [0 1> because T may not

be [0 1> for the negation body to be satisfied, negation succeeded over [0 1>. So a tuple p(T:[0 1>) issued before program execution starts.

STEP 1:
Since there is no positive clause body, the execution of tuple p does nothing.

```
STEP 2
==========================================================================
tuple list              derived                 pending list
--------------------------------------------------------------------------
current-time: 0
  p(T:[0 1>)            negation partially    =>p(T:[1 w>)<-not(p(S:[0 w>)
                        failed on [1 2>.                        T=S+1).
--------------------------------------------------------------------------
```

Execution reached the end of tuple list and next time becomes INFINITE, it is time to evaluate pending tuple. When evaluating the negation part of the pending tuple, tuple p(T:[0 1>) as input is unified with p(S:[0 w>) causing the value of S to be "squeezed" to [0 1> and resulting T:[1 2>, so negation body satisfied over time interval [1 2>, i.e. negation is partially failed over this range, no tuple could be drawn for now, pending tuple reduced to time stamp [2 w> and time stamp S for p inside negation became [1 w> as the result of time range calculation (S = T:[2 w> - 1). The reduced pending tuple is succeeded for now but has to remain in pending waiting for future evaluation.

The partial failure pattern for this case (refer to Figure 1):

```
  failed   succeeded
1        2            w
|========|------------
Tpl      Tpu'        Tpu
Tpl'
```

```
STEP 3
==========================================================================
tuple list              derived                 pending list
--------------------------------------------------------------------------
current-time: 0
  p(T:[0 1>)            negation partially    =>p(T:[2 w>)<-not(p(S:[1 w>)
                        failed on [3 w>.                        T=S+1).
                        tuple drawn:
                        p(T:[2 3>)
--------------------------------------------------------------------------
```

No new tuple generated and next time remains INFINITE, continue to evaluate negation for remaining pending tuple;

No tuples could be found that satisfy p(S:[1 w>), negation succeeded;

A circle path found: tuple p derived from the pending tuple could directly be input to its own negation body and conditions for Critical Phase met;

Critical Phase entered:

    preliminary tuple p(T:[2 w>) issued tentatively,
    no partial execution involved for this particular case,
    negation re-evaluated with tuple p(T:[2 w>) as input,
    negation partially succeeded over time interval [2 3> (because S
    is [2 w>, T must be [3 w> for negation to fail),
    pending tuple reduced to time stamp [3 w>, it is failed for now
    but has to remain in pending list waiting for further
    investigation;

Correct tuple (p [2 3>) generated after Critical Phase.

The partial failure pattern for this case:

```
succeeded    failed
2        3            w
|--------|============
Tpl      Tpl'        Tpu
                     Tpu'
```

```
==========================================================================
tuple list             derived                 pending list
--------------------------------------------------------------------------
current-time: 2
  p(T:[0 1>)                                   p(T:[3 w>)<-not(p(S:[2 w>)
=>p(T:[2 3>)           nil                                     T=S+1).
--------------------------------------------------------------------------
  ...                  ...                     ...
```

```
program execution goes on repeatedly in exactly the same pattern as that
for step 1 through step 3. And the following sequence of tuples are to
be generated:

      p(T:[0 1>)
      p(T:[2 3>)
      p(T:[4 5>)
      p(T:[6 7>)
        ...
```

5 Conclusion

The algorithm described in this report has been implemented in Scheme language and tested on some non-trivial programs. While some informal reasonings have be included, the soundness and completeness of the algorithm still await a formal proof.

The strict time stratification and the separation of negation evaluation from the execution of positive part of clauses have made this algorithm much more efficient than previous one. The conservative approach for negation evaluation adopted in this algorithm gives rise to issues that are common in distributed computing as Starlog is inherently of parallel nature. That is the problem of "deadlock", a way must be found to break it down. The notion of **Critical Phase** and **Path Analysis** has been introduced to deal with such situations as *cyclic inter-clause dependency* and *future-dependency* in the negation evaluation. To some extent, the semantics aspect of Starlog programs has been involved. This makes it even more different to prove that the algorithm is sound and complete.

6 Acknowledgments

This work was completed with support from the National Science and Engineering Research Council of Canada. We would also like to thank Alan Dewar for his help.

References

[1] Cleary, J.G. "*Logical Arithmetic*" 1987.
Future Computing Systems 2, no.2, pp. 125-149.

[2] Cleary, J.G. "*Colliding Pucks Solved in a Temporal Logic*" 1990.
Proceedings Distributed Simulation Conference. San Diego, California, January.

[3] Klocksin, W.F. and Mellish, C.S. *Programming in Prolog.* 1981.
Springer-Verlag.

[4] Dybvig, R.K. *The SCHEME Programming Language.* 1987.
Prentice-Hall.

[5] Kaushik, V.N. *Starlog: from Semantics to Interpretation.* 1991.
MSc. Thesis, University of Calgary.

[6] Lloyd, J.W. *Foundations of Logic Programming.* 1987.
Springer-Verlag.

[7] Naqvi, S. and Krishnamurthy, R.
"*Database Updates in Logic Programming*" 1988.
ACM Symposium on PODS, pp. 261-262.

[8] Warren, D.S. "*Memoing for Logic Programs*" 1992.
Communications of the ACM, March 1992/Vol.35, No.3, pp. 94-111.