Time-ordered Forward Deduction
With Negation Evaluation Strategy Revised

Zhonge Xiao
John Cleary
Brian Unger

January 12, 1995

Contents

Abstract

This is the third report on "Time-ordered Forward Deduction" algorithm for Starlog. The negation evaluation part of the algorithm has been modified extensively based on experiments with previous implementations and further study. The new algorithm is optimistic in terms of its negation evaluation strategy. The notion of ***Open Tuple*** *and related* ***Negation Failure Bound Splitting (NFBS)*** *method are introduced. A complete definition of the new algorithm is given, it treats the execution of a program in a more consistent and uniform manner and does not involve any semantic aspects of a program. It is our believe that a formal proof of the algorithm is not too difficult.*

1 Overview

The previous approach for the negation evaluation in the *Time-ordered Forward Deduction* algorithm, which is described in the first report on the subject, is essentially conservative and becomes quite complicated in trying to resolve the **negation evaluation** problem in a *bottom-up* method for *Starlog*, a temporal logic programming language. Another concern has been that such a complicated algorithm involving some semantic aspect of the language is very hard to prove theoretically. Effort has been made to reduce the complexity and to design a more consistent method for the negation evaluation, the key issue in a bottom-up strategy.

The main changes to the previous approach involve two aspects:

- **negation conversion**
  Each **negated clause** is to be converted to a clause with a single **negated atom** plus one or more **negation clauses** (one **negation clause** for each negation part of the original clause). The **negated atom** in the new **negated clause** calls the new formed **negation clauses**. The arguments of the **negation clause** consist of all the variables that appear in both the positive part and negative part of the original clause. When unification is applied to the **negated atom** and one of the **negation clauses**, variable binding between the positive part of a negated clause and its negative part can be correctly preserved.

This type of conversion makes it possible to evaluate negations in much the same way as the way the positive part of clauses are evaluated, that is by "*firing tuples*" or "*executing clauses with input tuples*". in terms of our bottom-up method. The tuples derived from **negation clauses**, called **negation tuples** will be generated in time order just like ordinary tuples. A definitive evaluation can then be made for the negative part, it becomes a single **negated atom** now, of a clause if the time stamp of the **negated atom** is earlier than `Tc`, the **current time**, since all facts (possible tuples) are all available by `Tc`.

- **Open and Pending Tuples**

The notion of "**pending tuples**" used in previous algorithm refers to those **tuples** that contain negations and are pending for further information to decide on their truth or falsity. This situation is largely due to the fact that negations are most effectively used to express such thing as "*this is true until something else occurs*" in logic languages. This implies that the truth or duration of truth of these pending tuples also depends on future facts not just the facts known up to now. But on the other hand, some other tuples that are dependent on these pending tuples can not be generated in time. In the situation of cyclic dependency, a kind of "*dead lock*" could arise because of tuples are being pending for the results of themselves. This is what has made the current implementation complicated in trying to break up subtle "*dead lock*" situations in the negation evaluation.

In addition to the **pending tuple** concept, a new notion called **open tuple** is introduced. **Open tuples**" are those that are derived from **pending tuples** and they remain "*open*" until they are *secure* or *safe*, then we say that they are closed. **Open tuples** simply mean that they are not *safe* they may well change as new generated tuples prove that this is necessary.
Tuples that have interval time value have the semantics that they are true only for certain period of time. For **open tuples**, once their lower boundary of time stamp, or start time become known, they will be issued immediately even though their upper boundaries are still dependent on future facts. They are "open" on their upper boundary. When future facts eventually arrive and their upper boundaries can be determined, the final tuple will replace the **open tuple** and those

derived from that **open tuple** will be replaced in turn or deleted in some cases. In some situations, a **open tuple** may have to split up to a tuple having definite interval time stamp and a new **open tuple**. A tuple with "*point*" time stamp is also considered "*open*" when it is first derived from a **pending tuple** as it may eventually be proven not true in situations where *cyclic dependency* of inter-clause relation exists.

One thing very nice about the notion of **open tuple** is that the cyclic dependency presents no more challenge. A posted **open tuple** can be fired and the results may in turn affect the **open tuple** itself. This is what **open tuple** is meant to do. The point here is that no "wait" involved, just go ahead and refine the **open tuple** as more facts show up until it becomes definite (**closed tuple**). In some sense, this resembles the issue of blocking and non-blocking in communication protocols.

The current implementation is indeed a conservative approach, a way must be found to break the "*dead lock*" situations which would arise inevitably. The approach being presented here is an optimistic one. With this optimistic approach, a more consistent algorithm can be worked out to cope, in a uniform fashion, with such situations as *future dependency* and *cyclic dependency*.

The framework of the revised algorithm could remain the same as that of current implementation. The negation evaluation part will have to undergo extensive modification. However, the techniques used are still applicable. To a large extent, re-implementation involves primarily re-organizing existing code. The result shall be a much simpler and sound one.

2 Summary of the Modified Method

The following is the summary of Time-ordered Forward Deduction, a bottom-up evaluation algorithm for Starlog with interval arithmetic, time, and negation.

1. Causality principle:

   ```
   Th >= Ti
   ```

   where `Th` is the time stamp of the head of a clause and `Ti` is the time stamp of any atom in the body including negation part. If the time stamp is of interval value, the lower boundary is used.

   Another condition is that in situations where *cyclic dependency* of inter-clause relations exists:

   > *any negations along the cyclic path must be accompanied by a finite time advance.*

2. The execution steps are marked with `Tc` (**Current Time**) which equals to the time stamp value (lower boundary if interval value) of the tuple being considered or involved in current step. Execution proceeds in non-decreasing `Tc` time order. Combined with (1) above, it is guaranteed that

   > *at `Tc`, no tuples with `Th < Tc` could ever be generated.*

3. The execution of a Starlog program is organized around R-trees (Rule Trees). A **rule** (original **clause**) and its derived **clause instances** and **tuples** including **open tuples** and **pending tuples**) are stored in an single well structured data object – R-tree. R-tree is not just a data structure for the storage of clauses and tuples, it is also a control structure for carrying out the execution within the rule.

4. **Negated rules** are rewritten into a new **negated rule** and, associated with it, one **negation rule** for each negation in its body.

The negation part of the new rule is reduced to a single atom, a **negated atom**. The predicate name of the **negated atom** is constructed by prefixing the rule's predicate name with a *underscore* character. The **negation rules** actually define the predicate which the **negated atom** calls on.

Each **negation rule** takes the original negation part respectively as its body. The arguments of a **negation rule** consist of those variables that appear in both the positive part and the negation part of the original rule.

multiple **negation rules** may exist to define the **negated atom** of a rule. These **negation rules** are treated like ordinary rules in terms of their execution except that the tuples derived from these rules, called **negation tuples**, are passive, that is they are used only for *unify* when evaluating the **negated atom**. They are, however, essential in negation evaluation.

5. The evaluation of a **negated rule** is carried out only when:

   (a) *a negated clause instance has become a* ***pending tuple****.*

   (b) *the time stamp of the* ***pending tuple*** *equals to* `Tc` *or* `Tc` *is about to advance through its time stamp.*

   The evaluation of a **negated atom** is to verify its truth or falsity by trying to unify it with currently valid **negation tuples** associated with it. These **negation tuples** are kept in a list (**negation tuples list**) which is associated with the **negated atom** of a **negated rule**.

6. **Pending tuples** will be evaluated and the results of evaluation issued as **open tuples** if their time stamp or the lower boundary of time stamp equals `Tc`.

   As a **open tuple** is not "*safe*" because future facts may affect the result of its negation part, therefore the very truth of itself. This is quite different from the **non-negated rules**. Tuples derived from these rules are definite because they are based on a set of known facts and only this set of facts. With negation, we are trying to prove based on all the relevant facts and assuming there were no more unknown facts.

7. The *time range* of a **open tuple** may change when new **negation tuples**, which is associated with it, are generated after its issuing, or when any changes happen to existing **negation tuples**. The resulting consequences from firing a **open tuple** must also be re-examined (or **re-visited**) if the *range* of the **open tuple's** time stamp ever changes. Proper techniques have to adopted to deal with subtle situations such as *cyclic dependency*. One possible implementation could be to mark the execution path of a **open tuple** and to undo the executions if the **open tuple** changes.

8. Once a **tuple** (not a **open tuple**) is executed or used, its possible consequences have all been derived. They are no longer needed and can be garbage collected. But, a "*used*" **negation tuple** does not become obsolete until `Tc` have advanced to a future time greater than its time stamp (upper boundary if of interval value).

9. A **open tuple** will become fixed or "*closed*" when `Tc` advances through its time (`Tc` greater than its upper boundary if of interval value).

10. It is common that a **pending tuple** may succeeds over part of the range of its time stamp and it becomes necessary to split the **pending tuple** up. A **NFBS** method is developed as to when and how a **pending tuple** with interval time value should split up as this is essential to complete a negation evaluation algorithm.

3 Definition of the Algorithm

In this section a more strict definition of the algorithm for the **Time-ordered Forward Deduction** method is given. The algorithm must be general enough to be applicable to all cases and be used as guide lines for implementation. It also has to be delicate to be able to handle subtle situations, specifically in the negation evaluation, in a uniform and consistent manner.

1. **Basic Execution Control**

   Each step of execution uses a **tuple** or a **pending tuple**, called the **Current Tuple**, and is marked with the **Current Time** `Tc`:

   > *`Tc` is set to the time stamp value (lower boundary, if interval value) of the **Current Tuple**.*

   The criteria for selecting the **Current Tuple** are:

   (a) *select a **tuple** or **pending tuple** so that the `Tc` will be the lowest possible.*

   (b) *a **pending tuple** will be selected only if there are no more **tuples** with time stamp value less than or equal to that of the selected pending tuple.*

   (c) *if there is more than one **tuple** with the same time stamp value, which one gets executed first does not matter.*

   (d) *if there is more than one **pending tuple** with the same time stamp value, it would be better to select one randomly so that the cost of possible re-visiting (back-tracing) in negation evaluation could be averaged. A more sophisticated selecting method could be employed, for example, analyzing the pattern of inter-links of these pending tuples.*

   The execution of each step consists of executing a set of R-trees. These R-trees contain at least one atom in their body that has the same predicate name as that of the **Current Tuple**. This includes the R-trees of **negation rules**.

   Execution terminates when there are no more **tuples** or **pending tuples** left.

2. **Execution of R-trees**
   The execution of a R-tree starts from the root where the original rule is kept, the derived clause instance (if derivable) is stored one level down the tree in the **home node** of the atom being "*unified*" with the **input tuple**, the **Current Tuple**, (each node is associated with an **non-primitive atom** in the rule's body), then goes to other sibling nodes which will be treated as sub-trees. This process is recursive until it comes to a leaf node where derived **tuples** or **pending tuples** are put onto list.

   The main advantage of guiding the execution by an R-tree structure is that it is much easier to keep trace of execution which is very important for the "*revisiting*" process in the case that an **open tuple** turns out not to be "*safe*" (not true, or partially true).

3. **Conversion of Negated Rules**
   Rules that contain one or more negated goals are rewritten to contain a single **negated atom** and, associated with it, a system-created **negation rule** for each negation in its body. The rules for the conversion are as follows:

   (a) *The new rule is the same as original one except that its negation part is collapsed into a single atom with the same predicate name as the* ***negation rule****.*

   (b) *The* ***negation rule*** *takes the negation part of the original rule as its body, its arguments consist of all the variables that appear in both the original negation part and the positive part plus a dummy time stamp. Besides, extra constraint primitives are added to its body which essentially specify that the value of a dummy time stamp is not less than any other time stamps taken as arguments from the original rule. If there is only one argument is a time stamp, it can be used, instead of a dummy one, as the time stamp of the* ***negation rule****.*

   (c) *The predicate name of the* ***negation rule*** *is formed by adding an underscore character as prefix to the name of the original rule.*

   multiple **negation rules** may exist to define the **negated atom** of the new rule if there are more than one negation in the original rule.

These **negation rules** are treated like ordinary rules in terms of their execution except that the tuples derived from these rules, called **negation tuples**, are passive, that is they are used only for *unify* when evaluating the **negated atom**. **Nested negations** are possible.

4. **Control of Negation Evaluation**
   The negation part of clause instances derived from **negated rules** during execution are not to be evaluated until:

   (a) *it becomes a **pending tuple**, that is, it contains only primitives and single negated atom.*

   (b) *the time stamp of the **pending tuple** is equal to* `Tc`*, or* `Tc` *is about to advance through the value of its time stamp.*

5. **Constructive Negation Evaluation**
   The truth or falsity of a **negated atom** is verified by trying to unify it with currently valid **negation tuples**. These **negation tuples** are kept in a list, the **negation tuples list** which is associated with the **negated atom**. The following steps are involved:

   (a) *Unify the **negated atom** with a **negation tuple** in the **negation tuples list** and propagate the new varible bindings, resulted from the unification, to the rest of the **pending tuple**.*

   (b) *Re-execute the primitives of the **pending tuple**, if there are any primitives. Suppose the resulting time stamp for the **pending tuple** is* `Tnti`*, then the negation will fail at* `Tnti` *or over* `Tnti` *(if of interval value).*

   (c) *If more than one **negation tuple** exist in the list, repeat step (a) through step (b) for each of them.*

   (d) *Let the original time stamp of the **pending tuple** under evaluation be Tph, a **tuple** with time stamp Tth can then be drawn (or the **pending tuple** is true over Tth), Tth can be obtained from:*

   $$Tth = Tph \cap \neg(Tnt1 \cup Tnt2 \cup ...Tntj)$$

   where j is the number of **negation tuples** in the **negation tuples list**. Tth could be disjoint.

This is a constructive strategy in the sense that the **negated atom** of a **pending tuple** actively participates in variable binding like ordinary atom. In fact, the *falsity* or *failure* is guaranteed by the existence of a **negation tuple**, so the similar procedure as that of deriving a ordinary tuple is being used in the evaluation of **negated atoms** of pending tuples except for the resulting tuples is opposite.

6. **Open Tuples** (*"unsafe" tuples*)

   A **tuple** drawn from a **pending tuple** is an **open tuple**, or **unsafe tuple**, and it will remain *open* or *unsafe* until `Tc` advances past its time (`Tc` greater than upper boundary if interval time), or a stronger condition is met (see (9) below).

   For the purpose of *revisiting*, the **pending tuple** which has yielded the **open tuple** is reserved and associated with the **open tuple**.

7. **Negation Failure Bound Splitting**

   The absence or existence of **negation tuples** are the very cause of the truth or falsity of a **pending tuple**. *Failure* caused by a **negation tuple** is definite as long as the **negation tuple** remains true, and it will not be affected by other **negation tuples**, while *success* is temporary as the arrival of new **negation tuples** may cause more *failures*. Each *failure* takes away a piece from a **pending tuple** or even claims the whole of it. The time range of a **pending tuple** will be split into parts when and only when:

   > *the result of negation evaluation for a* ***pending tuple*** *is partially true, or failed over certain time range,*

   the **pending tuple** will then be split into a **pending tuple** (not "*pending*" for now) which is true on its entire time range (which is a portion of the original pending tuple's time interval) and therefore yields an corresponding **open tuple** on that time range, and another **pending tuple**, which is false on the rest of the original time interval and it is bound to the **negation tuple** that causes the *failure*. This situation may well change due to arrivals of new **negation tuples** or changes to exiting ones. Only when **current time** `Tc` advances through the time of a **open tuple** or a **negation tuple**, will the tuple (either **open**

tuple or **negation tuple**) become "*fixed*" and the pending tuple associated with it be collected away.
Let Tot be the time of the **open tuple**, Tnt the time of the failed **pending tuple**, and Tph be the time stamp of the **pending tuple** before splitting, then

$$Tph = Tot \cup Tnt$$

Notice that each **open tuple** has a *successful* **pending tuple** assocaited with it while each *failed* **pending tuple** is caused by a **negation tuple**. Multiple *failed* **pending tuples** may exist, each is linked to a **negation tuple** that has caused its failure.

8. **Revisiting**

Revisiting refers to the process of *re-evaluating a previous* ***pending tuple*** *and re-considering the consequences of the* ***open tuple****, if any, previously derived from it.*

Revisiting is initiated when:

(a) *a new* ***negation tuple*** *is generated.*

(b) *a existing* ***negation tuple*** *in the list has to be changed or deleted due to the* ***open tuple****, which generated the* ***negation tuple*** *previously, has been changed or deleted.*

In case (a), all existing **open tuples** have to be re-considered by re-evaluating **pending tuples** associated with them. Practically re-evaluation will be carried out for a previously *succeeded* **pending tuple** only if the time of the new arrived **negation tuple** falls into the range of that **pending tuple**. If under the new situation the old **open tuple** is not true or only partially true, then the consequences of its previous execution has to be deleted, or its previous execution path be re-visited.

In case (b), only the **pending tuple** which is associated with the changed **negation tuple** has to be re-evaluated and the re-evaluation needs to be done only for the new **negation tuple**. If the re-evaluation results in a *success* and the generation of a new **open tuple**, it must not be in conflict with other existing **negation tuples**. Any **tuple** derivable under the new situation is a new **open tuple** which has not been executed yet and subject to possible future changes.

If part of the time range of the changed **negation tuple** in case (b) falls out of the original time interval, the excessive part can be treated as a new **negation tuple** like case (a).

For re-evaluation, the same steps discussed in (5) apply.

9. **Conditions on the Closure of Open Tuples**

   The following conditions can be used to determine if a **open tuple** can be "*closed*" and its associated **pending tuples** deleted:

   (a) *`Tc` is greater than the time stamp value of the* ***open tuple*** *(upper boundary if interval time). This is a decisive condition since no* ***negation tuples*** *with earlier time than that of the* ***open tuple*** *can ever be generated which would affect the truth of the* ***open tuple***.

   (b) *A stronger condition is that `Tc` greater than the time (upper boundary if interval value) of the* ***negated atom*** *in the* ***pending tuple*** *from which the* ***open tuple*** *has been drawn, since no* ***negation tuples*** *with earlier time than `Tc` can be generated which would fall into the time range of the* ***negated atom***. *This condition is sometimes hard to meet, especially when a dummy time stamp is used for the* ***negation rule***.

Rules (1) and (2) define the execution of "*positive*" part, while rules (3) through (5) define the execution of negation. The last four are designed to cope with situations of *"cyclic dependency"* or situations where the time period of truth of a tuple depends on future facts, which are typical of *database applications*. The most distinctive feature of negation handling in this algorithm are the notions of **OP** (**Open Tuple**) and **NFBS** (**Negation Failure Bound Splitting**) , which does not count on any semantic aspect of user programs and therefore is a generic algorithm.

4 Conclusion

This new algorithm has been tried out, by hand tracing, on all the example programs that have been tested for current implementation of Starlog. It handles different cases in a more consistent and uniform manner. This new algorithm is much simpler and robust.

5 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.

A Example program - Oscillator

This is a very simple, but non-trivial program. It contains a single statement, a recursive rule which simulates the wave produced by an Oscillator. This example shows step by step how the Time-ordered Forward Deduction algorithm works in a general, consistent manner.

```
program:
                 p(T) :- not(p(S), T is S+1).


after rewrite:
                 p(T) :- not(_p(T)).
                _p(T) :- P(S), T is S+1.
```

```
execution:
=======================================================
start.

TUPLES       PENDING TUPLES                NEGATION TUPLES
  []         p(T:[0 W>):- not(_p(T))            []

-------------------------------------------------------
step 1. evaluating p(T:[0 W>):- , Tc = 0.

TUPLES        PENDING TUPLES               NEGATION TUPLES
p(T:[0 W>)<-- p(T:[0 W>):- not(_p(T))           []


succ. =============
      |------------|
      0            W
-------------------------------------------------------
step 2. executing p(T:[0 W>), Tc = 0.

TUPLES        PENDING TUPLES               NEGATION TUPLES
p(T:[0 W>)<-- p(T:[0 W>):- not(_p(T))     _p(T:[1 W>)


    new negation tuple generated.
-------------------------------------------------------
step 3. re-examine p(T:[0 W>):- , Tc = 0.

TUPLES        PENDING TUPLES               NEGATION TUPLES
p(T:[0 1>)<-- p(T:[0 1>):- not(_p(T))
              p(T:[1 W>):- not(_p(T)) --> _p(T:[1 W>)

succ. ===
      |--|---------|
      0  1         W
fail.    ==========
-------------------------------------------------------
```

```
-------------------------------------------------------
step 4. re-visit p(T:[0 W>) with p(T:[0 1>), Tc = 0.

TUPLES         PENDING TUPLES             NEGATION TUPLES
p(T:[0 1>)<-- p(T:[0 1>):- not(_p(T))
              p(T:[1 W>):- not(_p(T)) --> _p(T:[1 2>)


    existing negation tuple changed.
-------------------------------------------------------
step 5. re-examine p(T:[1 W>):- , Tc = 0.

TUPLES         PENDING TUPLES             NEGATION TUPLES
p(T:[0 1>)<-- p(T:[0 1>):- not(_p(T))
              p(T:[1 2>):- not(_p(T)) --> _p(T:[1 2>)
p(T:[2 W>)<-- p(T:[2 W>):- not(_p(T))


succ.    ==========
      |--|---------|
      1  2         W
fail. ===
-------------------------------------------------------
step 6. execute p(T:[2 W>), Tc = 2.

TUPLES         PENDING TUPLES             NEGATION TUPLES
p(T:[0 1>)<-- p(T:[0 1>):- not(_p(T))
              p(T:[1 2>):- not(_p(T)) --> _p(T:[1 2>)
p(T:[2 W>)<-- p(T:[2 W>):- not(_p(T))
                                          _p(T:[3 W>)


    new negation tuple generated.
-------------------------------------------------------
```

```
--------------------------------------------------------
step 7. re-examine p(T:[2 W>):-, Tc = 2.

TUPLES        PENDING TUPLES            NEGATION TUPLES
p(T:[2 3>)<-- p(T:[2 3>):- not(_p(T))
              p(T:[3 W>):- not(_p(T)) --> _p(T:[3 W>)


succ. ===
      |--|---------|
      2  3         W
fail.    ==========
--------------------------------------------------------
step 8. re-visit p(T:[2 W>) with p(T:[2 3>), Tc = 2.

TUPLES        PENDING TUPLES            NEGATION TUPLES
p(T:[2 3>)<-- p(T:[2 3>):- not(_p(T))
              p(T:[3 W>):- not(_p(T)) --> _p(T:[3 4>)


    existing negation tuple changed.
--------------------------------------------------------
step 9. re-examine p(T:[3 W>):-, Tc = 2.

TUPLES        PENDING TUPLES            NEGATION TUPLES
p(T:[2 3>)<-- p(T:[2 3>):- not(_p(T))
              p(T:[3 4>):- not(_p(T)) --> _p(T:[3 4>)
p(T:[4 W>)<-- p(T:[3 4>):- not(_p(T))


succ.   ==========
     |--|---------|
     3  4         W
fail.===
--------------------------------------------------------
step 10. execute p(T:[4 W>), Tc = 4.
```

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] Clocksin, 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.