The Event Calculus as a Linear Logic Program

TR95–24

Vladimir Alexiev *

University of Alberta, Dept of Comp Sci, 615 GSB, Edmonton, AB T5K 1A2, Canada.
<vladimir@cs.ualberta.ca>

Abstract. The traditional presentation of Kowalski's Event Calculus as a logic program uses Negation-as-Failure (NAF) in an essential way to support persistence of fluents. In this paper we present an implementation of Event Calculus as a purely logical (without NAF) Linear Logic (LL) program. This work demonstrates some of the internal non-monotonic features of LL and its suitability for knowledge update (as opposed to knowledge revision). Although NAF is an ontologically sufficient solution to the frame problem, the LL solution is implementationally superior. Handling of incomplete temporal descriptions and support for ramifications (derived fluents) are also considered.
Keywords: event calculus, linear logic, negation as failure, knowledge update.

1 Introduction

The Event Calculus (EC) of [Kowalski and Sergot, 1986] is a theory of events (actions) and the fluents (predicates) that they precipitate. An important property of the theory is that it is rendered as a logic program, and is thus executable. Negation-as-Failure (NAF) plays an important role in this program, ensuring the persistence of fluents through time until the occurrence of an event that terminates them.

Historically, EC has been an antithesis of another theory of actions, the Situation Calculus (SC) [McCarthy and Hayes, 1969]. The SC adopts as primitive the notion of *situation*, the set of all fluents that hold between two events. Events map one situation to another, the result of applying the event in the situation. The EC purports to make the global notion of situation unnecessary, by letting fluents evolve "independently" in time. This solves the Frame problem: the necessity to reason explicitly about the persistence of fluents between situations. However, EC's use of NAF for persistence somewhat undermines this effort, by presenting both representation and implementation problems (Section 3.1).

Linear Logic (LL) [Girard, 1987] is rapidly gaining popularity and applications in Computing Science [Alexiev, 1994], mainly due to its interpretation of formulas as resources and not as timeless properties. This feature of LL makes it possible to account for change in a clean, logical and simple way.

This paper purports to use the resource-consciousness of LL to reimplement EC in a purely logical way and thus overcome the problems related with NAF. We hope that this paper delivers on some of the promises of LL for applications in AI where change is essential. The rest of the paper is as follows: Section 2 introduces the two LL programming languages that we use, Lolli and Lygon. Section 3 describes a simplified version of EC, implements it in LL, and proves a correspondence between the original definition and the LL implementation. Section 4 introduces several extensions to SEC and presents considerations how they can be accommodated in LL. Finally, Section 5 contains some concluding remarks.

2 Linear Logic Programming

We implemented our LL theory of EC in two different LL programming languages, because each of them contains some features that are not present in the other. Although similar in spirit and genesis, they are quite different in detail. We introduce them briefly below.

* The author gratefully acknowledges the support of a Killam Memorial Scholarship

2.1 Lolli

The language Lolli[2] was introduced by [Hodas, 1992; Hodas and Miller, 1994]. It is based on an intuitionistic fragment of LL and Miller's earlier work on λProlog (and more generally, Uniform Proofs). Accordingly, Lolli is higher-order and handles only single-conclusion sequents. Lolli is implemented in SML and is quite effective. A simple module system is provided, however no debugger is available at present. The fragment of LL that Lolli supports is:

$$
\begin{aligned}
R &::= A \mid \top \mid R \& R \mid A \circ\!\!- G \mid A \Leftarrow G \mid \forall X.G \\
G &::= A \mid \top \mid G \& G \mid R \multimap G \mid R \Rightarrow G \mid \forall X.G \\
&\quad \mid \mathbf{1} \mid G \otimes G \mid G \oplus G \mid\ !G \mid \exists X.G \mid G \rightarrow G | G
\end{aligned}
$$

The concrete syntax used by Lolli is as follows:

$\otimes$	$\parr$	&	$\oplus$	$\circ\!\!-$	$\multimap$	$\Leftarrow$	$\Rightarrow$	!	?	$\forall$	$\exists$	**1**	$\bot$	$\top$	**0**	$(\cdot)^\perp$	if-then-else
,		&	;	:-	-o	<=	=>	{·}		forall	exists	true		erase			·->·|·

2.2 Lygon

The language Lygon[3] was introduced by [Harland and Winikoff, 1995; Winikoff and Harland, 1994] based on earlier work of [Harland and Pym, 1994; Harland and Pym, 1992]. It is based on classical LL (and so works with multiple-conclusion sequents), but is essentially first-order.[4] Lygon is implemented as a meta-interpreter over BinProlog. An unsophisticated debugger is provided. The fragment of LL that Lygon implements is:

$$
\begin{aligned}
D &::= \exists X \forall Y.D' \mid D \otimes D \\
D' &::= A \mid A \circ\!\!- G \\
G &::= \mathbf{1} \mid \bot \mid \top \mid \mathbf{0} \mid A \mid A^\perp \mid G \otimes G \mid G \parr G \mid G \& G \mid G \oplus G \mid \exists X.G \mid\ !G \mid\ ?G^\perp \mid \texttt{once}\ G
\end{aligned}
$$

Lygon has a richer goal sub-language than Lolli, but a poorer clause sub-language. The paper [Harland *et al.*, 1995] proposes a richer clause language (including integrity constraints $G \multimap \bot$), but the current implementation supports only the fragment listed above.

The concrete syntax used by Lolli is as follows:

$\otimes$	$\parr$	&	$\oplus$	$\circ\!\!-$	$\multimap$	$\Leftarrow$	$\Rightarrow$	!	?	$\forall$	$\exists$	**1**	$\bot$	$\top$	**0**	$(\cdot)^\perp$	commitment
*	#	&	@	<-				!	?	forall, quant	exists, query	one	bot	top	zero	neg ·	once

3 Simplified Event Calculus

We first consider the simplified version of EC (SEC) [Kowalski, 1992], which historically was more widely used. In subsequent sections we consider extensions of SEC. For simplicity's sake, we assume that the occurrence times of all events are known, or equivalently, that the events are totally ordered. It would not be hard to adapt our results for the case of partial ordering of events, but this is not the emphasis of the present work.

One possible presentation of SEC as a Prolog program is:

[2] http://www.cs.hmc.edu/ hodas/research/lolli/

[3] http://www.cs.mu.oz.au/ winikoff/lygon/lygon.html

[4] One can use the builtin call of the underlying Prolog system though.

```
holds(F,T) :-
  happens(Ei,Ti), initiates(Ei,F),
  happens(Et,Tt), terminates(Et,F),
  between(Ti,T,Tt), not broken(Ti,F,Tt).
broken(Ti,F,Tt) :-
  happens(E,T), between(Ti,T,Tt),
  (initiates(E,F1); terminates(E,F1)),
  exclusive(F,F1).
between(T1,T2,T3) :-
  T1<T2, T2<T3.
exclusive(F,F).
```

(here we use mnemonic variables of the form E: Event, F: Fluent, T: Time point). Please note the essential use of NAF (`not broken`) to implement the default persistence of the fluent between event occurrences. The "initial conditions" of a narrative and the possibility of non-terminated fluents are accounted for by the following additional clauses:

```
initiates(start,F) :-
  initially(F).
happens(start,0).
holds(F,T) :-
  happens(Ei,Ti), initiates(Ei,F),
  Ti<T, not broken(Ti,F,T).
```

The generic theory above is complemented with a domain-specific theory of the form

```
initially(unloaded).        initially(alive).
initiates(load,loaded).     terminates(load,unloaded).
% the following two facts will be qualified later with a precondition holds(loaded)
initiates(shoot,dead).      terminates(shoot,alive).
initiates(shoot,unloaded).  terminates(shoot,loaded).
exclusive(alive,dead).      exclusive(loaded,unloaded).
happens(load,1). happens(wait,2). happens(shoot,3).
```

3.1 Shortcomings of NAF

SEC as formulated above is an acyclic program[5] [Provetti, 1994, Sec. 4.3], therefore all major semantics for logic programs with NAF agree for the SEC. However, we still see several problems with the use of NAF in EC, some representational and some implementational.

- Some implementations of the Frame axiom through NAF suffer from the following form of over-commitment: if a fluent F' is derived from a fluent F then default persistence may cause F' to persist in time even after its base F is terminated. This problem is not present in the above presentation of SEC, at least for domain theories in which base and derived fluents are separated in the sense that no derived fluent occurs in a `initiates` or `terminates` clause. See Section 4.2 for further discussion.
- In the extension of SEC with incomplete information (see Section 4.3), NAF leads to over-committed conclusions that certain fluents do not hold. Consider this example (from [Pinto and Reiter, 1993]):

  ```
  happens(e1,1).     happens(e2,2).
  initiates(e1,f1).  terminates(e2,f2).
  exclusive(f1,f2).
  ```

[5] But not a locally stratified one.

Clark's completion of the EC theory given in [Kowalski and Sergot, 1986] is able to infer the existence of an intervening event `e` that initiates `f2`, but for any `t` $\in [1,2]$, the query `holds(f2,t)` fails because there is no specific information about the occurrence of `e`. This flaw largely undermines the usefulness of the extended EC as described in [Kowalski and Sergot, 1986].

- The present implementation of temporal persistence through NAF is very inefficient. The predicate `broken` has to check all event occurrences, even the events that have nothing to do with the fluent at hand. The most constraining subgoal in the clause for `broken` is `exclusive(F,F1)`, which however cannot be used early in the clause because `initiates` and `terminates` are not necessarily exclusive. One solution to this problem is described in [Kowalski, 1992, p.138ff]: every fact `happens(E,T)` is executed forward a couple of steps to derive its `initiates` and `terminates` consequences, and these are stored as a cluster indexed by `F`. This allows fast retrieval of the events that are related to `F`. In fact the present paper implements more or less this solution, in a purely logical theory.

3.2 SEC in Lolli

The basic idea of our LL implementation of SEC is simple: first, we record the initial state of every fluent `F` as an atom `int(F,0,infty)` (`F` holds for an interval from 0 to infinity). These atoms are stored in the linear part of the execution context, so it is easy to delete and update them. Then an event occurrence `happens(E,T)` splits the validity intervals of all concerned events in two, the first part keeping the same truth value as the old interval, and the second part determined by the new event.

A Lolli program implementing this ideas is:

```
MODULE sec.
LOCAL init record insert.

start G :- initially Fs, init Fs G.
init nil G :- G.
init (F::Fs) G :- int F 0 infty -o init Fs G.

happens E T G :- causes E Fs, record Fs T G.
record nil T G :- G.
record (F::Fs) T G :- insert F T (record Fs T G).
insert F2 T G :- exclusive F1 F2, int F1 T1 T2, between T1 T T2,
    (int F1 T1 T -o int F2 T T2 -o G).
exclusive F F.
exclusive F (not F).
exclusive (not F) F.
between T1 T2 T3 :- less T1 T2, less T2 T3.
less T1 T2 :- T2=infty -> true | (T1=infty -> fail | T1<T2).

holds F T G :- (int F T1 T2, between T1 T T2, erase) & G.
holds F T :- holds F T erase.
holds F :- holds F infty.
prtint.
prtint :- int F T1 T2, write (int F T1 T2), nl, prtint.
```

Notes on the Lolli syntax:

- `MODULE` and `LOCAL` are declarations concerning the Lolli module system. A module is used by loading it with the following syntax: `M --o G`.
- We use the following mnemonic variable names: E: Event, T: Time point, F: Fluent, Fs: a list of Fluents, G: subsequent Goal (continuation).

- Lolli uses curried predicate syntax (no parentheses and commas).
- The language is higher-order, and this is why we do not have to wrap the variable subgoal `G` in a meta-predicate `call`.
- The double colon in `(F::Fs)` is cons (the list constructor), and `nil` is the neutral element of lists.

Notes about the program:

- We use extensively continuation-passing style. Most often the continuation `G` is simply passed to the subgoals, but sometimes more specialized treatment is required. For example the clause

  ```
  init (F::Fs) G :- int F 0 infty -o init Fs G.
  ```

  executes the continuation in a context amended with the atom `int F 0 infty`. As a matter of fact, the reason we used continuations is exactly because there is no other way to insert atoms in the linear context: Lolli does not support linear negation. The clause

  ```
  holds F T G :- (int F T1 T2, between T1 T T2, erase) & G.
  ```

  executes the continuation in an unchanged context, in other words it insulates the computation from the part enclosed in parentheses, or effectively makes that part behave as a simple test.
- `start` fetches the domain-theory atom `initially` (see below), starts the list-iterating predicate `init` and continues with the rest of the goal `G`.
- Every subgoal `happens E T` fetches the domain-theory atom `causes E Fs`, describing the effect of the event `E` and then iterates over the list of affected fluents `Fs` using `record`.
- `insert F2 T` consumes the recorded validity interval `int F1 T1 T2` of an exclusive fluent and splits it in two.[6] Before invoking the continuation, it inserts the two subintervals in the linear context.
- `exclusive` states that every fluent is exclusive with itself, and with its negation. The word `not` here is a simple data constructor and has nothing to do with NAF. For uniformity, we chose to represent the domain theory with axioms of the form

  ```
  causes e1 f.      causes e2 (not f).
  ```

  instead of the traditional

  ```
  initiates e1 f.   terminates e2 f.
  ```

- `less` implements a comparison operator that can deal with the special value `infty` (infinity). In its implementation we used the impure builtin `->`$\cdot|\cdot$ (if-then-else) because neither type-checking predicates (`number(T)`), nor disequality check, are available.
- `holds F T` checks if fluent `F` holds at time `T`. It has two variations, one that have a continuation, and another that checks at time infinity, which can be interpreted as the current time.
- Finally, o9`prtint` prints all accumulated validity intervals.

Complemented with a domain theory of the form

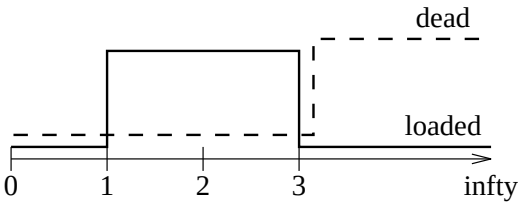
```
MODULE yale.

causes load (loaded :: nil).
causes unload (not loaded :: nil).
causes shoot (not loaded :: dead :: nil).  % will be qualified with holds(loaded)
causes wait nil.
initially (not loaded :: not dead :: nil).
```

the above program can answer queries such as

[6] We can assume that there is only one such interval, because there cannot be two exclusive fluents holding at the same time.

```
?- sec --o yale --o start (happens wait 2 (happens shoot 3 (happens load 1 prtint))).
int (not loaded) 0 1
int loaded 1 3
int (not loaded) 3 infty
int (not dead) 0 3
int dead 3 infty
solved
```



The output above signifies that `loaded` holds between times 1 and 3, and that `dead` holds from 3 to infinity. The events are specified in the goal instead of in the domain theory in order to activate forward-chaining over the `causes` facts. Please note that events do not have to be specified in chronological order, and that `wait` does not cause the problem described in [Hanks and McDermott, 1987] that plagues circumscription-based implementations of the Situation Calculus.

3.3 SEC in LYGON

As motivated in Section 2, we implemented SEC in two different LL programming languages, because neither of them gives us the full flexibility that we need. The LYGON program is similar to the LOLLI program:

```
start <- initially(Fs) * init(Fs).
init([]) <- bot.
init([F|Fs]) <- neg int(F,0,infty) # init(Fs).
happens(E,T) <- causes(E,Fs) * record(Fs,T).
record([],T) <- bot.
record([F|Fs],T) <- insert(F,T) # record(Fs,T).
insert(F2,T) <- exclusive(F1,F2) * int(F1,T1,T2) * between(T1,T,T2) *
  (neg int(F1,T1,T) # neg int(F2,T,T2)).
exclusive(F,F).
exclusive(F,not(F)).
exclusive(not(F),F).
between(T1,T2,T3) <- less(T1,T2) * less(T2,T3).
less(T1,T2) <-
  prolog(number(T1)) * prolog(number(T2)) * lt(T1,T2)
  @ prolog(number(T1)) * eq(infty,T2).
eq(X,X).
holds(F,T) <- int(F,T1,T2) * between(T1,T,T2) * top.
holds(F) <- holds(F,infty).
prtint.
prtint <- int(F,T1,T2) * print(int(F,T1,T2)) * nl * prtint.
```

The main differences are that in `init` and `insert` we use linear negation `neg` to insert the atoms `int` in the linear context, and in `less` we use the underlying BINPROLOG system for the predicate `number`.

Complemented with a domain theory of the form

```
causes(load, [loaded]).
causes(unload, [not(loaded)]).
causes(shoot, [not(loaded), dead]).
causes(wait, []).
initially([not(loaded), not(dead)]).
```

our program can answer queries of the form

```
?- start#happens(wait,2)#happens(shoot,3)#happens(load,1)#prtint.
int(not loaded,0,1)
int(loaded,1,3)
int(not loaded,3,infty)
int(not dead,0,3)
int(dead,3,infty)
Succeeded.
```

3.4 Faithfulness of the Proposed Implementation

In this section we prove that our implementation of SEC in LL is sound and complete with respect to the original definition of SEC, *i.e.* that they produce the same answers.

We first prove the property informally stated at the end of Section 3.2.

Lemma 1 (Order-independence). *Assuming complete initial information (every fluent or its negation is specified in the* `initially` *fact, the answers of the* LOLLI *program of Section 3.2 do not depend on the order in which* `happens` *atoms are given in the goal.*[7]

Proof. Let's consider an arbitrary fluent `F`. At any time the set of validity interval atoms about `F` forms a non-overlapping cover of the interval `[0..infty]`. This is proved by an easy induction: initially the linear context contains exactly one such interval, either `int F 0 infty` or `int (not F) 0 infty`. Let's consider the effect that a fact `happens E T` has on `F` (assuming that `E` affects `F`). It splits the unique interval `[T1..T2]` that contains `T` in two, `[T1..T]` and `[T..T2]`, even if the two parts assert the same truth value for `F`. Furthermore, the final set of such splittings does not depend on the order the splittings were done. □

Theorem 2 (Faithfulness). *The* PROLOG *program of Section 3 and the* LOLLI *program of Section 3.2 with corresponding domain theories give the same answers to queries of the form* `holds(F,t)` *where* `t` *is bound and does not coincide with any of the event occurrence times.*

Proof. Given the previous lemma (and the trivial fact that the PROLOG program does not have an "order of events"), we can assume that the events are described in chronological order. Now we can perform a proof by induction on the number of events.

Base Case If no events are recorded, both programs answer the query `holds(F,t)` with the fluents `F` that are mentioned in `initially`, independent of the concrete time `t` (provided it is positive).

Induction Step Assume that the claim holds for a sequence of events and let's append an event `holds(E,T)` at its end. The LOLLI program splits the last interval (which extends to infinity) of every affected fluent in two, the first part retaining the old truth value, and the second part assuming the new truth value. Therefore for any `t<T` the answers will not be affected by the new event, while for `T<t<infty` the answers correspond simply to the fluent list in `causes E Fs`.

Now let's turn to the PROLOG program. A bit subtler analysis is required because of the interference of the two `holds` clauses (for an internal interval and for the last interval). because in the PROLOG program the new event does not enable any derivations

□

4 Extensions to SEC

In this section we extend the SEC program described above in various ways, the last of which (Section 4.3) provides for most of the flexibility of the original EC [Kowalski and Sergot, 1986]. Other conceivable extensions of our approach which are not treated in this paper include:

[7] Of course, the answers do depend on the times specified in the `holds` atoms. Also, this lemma will not be valid when we introduce preconditions to `causes` predicates.

– Partially-ordered events.
– Branching time-line and hypothetical events. This would require the maintenance of a more complex interval structure in the linear context than the simple set of "arrays" we use now.

4.1 Preconditions

Many events are preconditioned on fluents that must hold for the event to be applicable. In fact the Yale shooting example as stated above (Section 3.2) does not take into account the domain knowledge that only shooting a loaded gun is effective. Here we show how preconditions can be added to our formalism.

First of all, we assume that the events are recorded in their chronological order, because otherwise a precondition `P` of an event `E2` depending on a previous event `E1` will not be satisfied if `E1` is recorded after `E2`. This limitation can be lifted by recording all events in a chronologically-ordered list and then "replaying" them for every query. However such a solution would defy the very spirit of our LL approach, namely that the consequences of events are computed as soon as possible and queries simply lookup this cached information.

To implement this sequencing of events, we found continuations very useful. The (traditional) connectives of LL are commutative (except for $\multimap$), therefore one should not use them to represent sequentially occurring events.[8] Therefore, we drop Lygon at this point and present this extension in Lolli.

There are several conceivable approaches to implementing preconditions:

1. Preconditions can be "sampled" at the current time, *i.e.* the time of the event concerned.
2. Preconditions can be sampled at time infinity. In light of the previous remark (the event time being the latest recorded time point), this is equivalent to 1.
3. Preconditions can be sampled at arbitrary times (hopefully somehow related to the current time). This approach allows the greatest flexibility, but it requires passing the current time to the `causes` facts. This can be achieved using quantification in Lolli.

As an example we implement the approach 2. First, we change the relevant domain clause from

```
causes shoot (not loaded :: dead :: nil).
```

to

```
causes shoot (precond dead (holds loaded) :: not loaded :: nil).
```

Please note that we had to change the list order of the two results (`not loaded` and `dead`), because if `not loaded` was recorded first, it would render the precondition of `dead` false. To conform to the notion that all results are effected simultaneously, one could adopt approach 1, and postulate that validity intervals include their endpoints.

We also change one of the clauses for `record`:

```
record (F::Fs) T G :-
  F=(precond F1 Pre) -> (Pre -> insert F1 T (record Fs T G)
                              | record Fs T G)
                      | insert F T (record Fs T G).
```

This involves using the impure operator if-then-else to recognize the type of list element (one with or without a `precond`), however this use has nothing to do with temporal information. In the clause above, if the element does not have a precondition or the precondition is satisfied, the element is inserted; else `record` just continues with the end of the list.

[8] In fact Lygon does not take into account the ordering of goals in the text of a program. For example a query `a#b#c` is run in the order `b,c,a`.

4.2 Ramifications

Adding ramifications (derived fluents) does not pose any problem to our approach. Furthermore, ramifications can be involved in `causes` facts.[9]. For example, if we extend our domain theory with

```
holds alive T G    :- holds (not dead) T G.
holds unloaded T G :- holds (not loaded) T G.
```

then we can reason about `alive` on the same footing as its base fluent `dead`.

If we also add

```
exclusive alive dead.
exclusive unloaded loaded.
holds dead T G    :- holds (not alive) T G.
holds loaded T G :- holds (not unloaded) T G.
```

and the general clauses

```
exclusive F (not F1) :- exclusive F F1.
exclusive (not F) F1 :- exclusive F F1.
```

then we can assert `causes` facts involving either of the fluents at will.

Accommodating fluents that are not related so directly (one being the negation of the other) is slightly more difficult. For example we could add a fluent `peaceful` with the clause

```
holds peaceful T G :- holds (not loaded) T G, holds alive T G.
```

However, the involvement of such fluents in `causes` is more problematic. If we assert that an event causes `peaceful`, how should this be interpreted, as the event causing both `not loaded` and `alive`? What about a derived fluent being the disjunction of two others, or an even more complex formula? Since there is no direct correspondence between logic connectives and the combinators of a logic of actions (*e.g.* Dynamic Logic [Harel, 1979]), the best approach seems to be to limit such derived fluents to the role of observations.

4.3 Handling of Incomplete Information

The original EC [Kowalski and Sergot, 1986] can handle several cases of incompletely specified events, and infer the existence of unspecified events from constraints imposed by specified events. Assume the domain theory

```
initiates(give(X,Y,Z), has(Z,Y)).  terminates(give(X,Y,Z), has(X,Y)).
exclusive(has(X,Y), has(Z,Y)) :- not X=Z.
happens(give(bob,book,mary),1).     happens(give(john,book,jim),2).
```

(here `give(X,Y,Z)` means "person `X` gives the object `Y` to person `Z`"). Then it can be inferred from the Clark's completion of the EC program that an intervening event of Mary giving the book to John (or even a chain of transfers of the book between persons) must have existed. However, due to the over-commitment caused by the use of NAF for temporal persistency, for every time `T` between 1 and 2, the program would answer that both Mary and John have the book. [Sadri and Kowalski, 1995] propose to solve this problem and make the framework generally more flexible through the use of positive programs amended with general integrity constraints.

One form of incomplete information is already handled by the LL programs in Section 3.2 and Section 3.3. Namely, since the effects of an event are recorded even if they already hold (due to the clause `exclusive(F,F).`), we can handle the case of an unspecified terminating event.[10] More complex cases can

[9] In the terms of deductive databases, the same predicate can be both EDB and IDB

[10] This same mechanism allows for event occurrences recorded not in the chronological order

be accommodated by allowing variables in the validity intervals recorded in the linear context. *E.g.* the example above can be handled by recording intervals[11]

```
int(has(mary,book),1,X).  int(not has(mary,book),X,infty).
int(has(john,book),X,2).  int(not has(john,book),2,infty).
```

(under an assumption of minimal number of unspecified transfers), where the variable time `X` serves to tie the fluents `has(mary,book)` and `has(john,book)` in an appropriate way. One will have to assert the appropriate chronological order by ordering the intervals sequentially in a data structure (*e.g.* list), but on more complicated cases a full temporal constraint system will be required.

Note A more systematic account of this problem will appear in the final version of this paper. More specifically, we are interested in investigating the expressiveness of our approach not simply on the examples of [Hanks and McDermott, 1987] and [Kowalski and Sergot, 1986], but in a more formal setting, such as the one given in [Gelfond and Lifschitz, 1993].

5 Concluding Remarks

5.1 Related Work

This work is in the general direction of applying LL to AI problems, especially ones where Non-Monotonic Reasoning is (or was thought to be) necessary. For a large class of problems only a limited form of non-monotonicity is sufficient, one which does not involve belief revision and database updates with complex formulas. For these problems LL seems to provide a natural purely-logical solution. Examples of such problems are conjunctive planning [Masseron *et al.*, 1993; Hölldobler, 1992; Jacopin, 1993] and hierarchies with exceptions [Fouqueré and Vauzeilles, 1994]. [Arima and Sawamura, 1993] argue that LL is appropriate for a logic of explanation and abduction.

The only work relating EC and LL that we are aware of is [Cervesato *et al.*, 1994]. An implementation of Modal EC in Lolli is presented. The Modal EC is a modification of EC which presumes all events and some partial ordering given in advance, and determines which fluents must necessarily hold or can possibly hold under all possible refinements of the ordering.

There is no deep relation between the present work and [Cervesato *et al.*, 1994]. It seems that the latter makes no essential use of the linearity properties of Lolli; in fact the implementation could have been done in Lolli's non-linear predecessor λProlog as well. NAF is used to provide temporal persistence in the same way as in [Kowalski, 1992]; `erase` is overused; and hypothetical refinements on the order of events (`beforeFact E1 E2`) are introduced in the non-linear part of the context (with intuitionistic instead of linear implication).

5.2 Future Work

Using LL context management provides for a simple purely-logical way of updating temporal information, and it seems preposterous to employ the heavy machinery of NAF for this purpose. However, NAF is still useful for more general patterns of deduction. The ability to check for the lack of certain information is useful. However, the result of this check should not be treated in the same way as the information it was based upon, or else paradoxes, conflicting extensions, or at least infeasible techniques, emerge. We are currently working on a "Logic of Objects and Properties" where some (linearly managed) formulas are regarded as "material objects" (resources) that can be produced, consumed, and transformed; and other formulas (classically managed) are regarded as properties of such objects. The absence of a certain object from the context (which we see as the main application of NAF) should be regarded as a property of the context.

[11] Together with the regular intervals `int(not has(bob,book),1,infty). int(has(jim,book),2,infty).`

Seemingly the Logic of Unity [Girard, 1993] will be useful for such a formalism, but we anticipate several differences, the most important being presence of two kinds of negation (object-level and property-level).

Similar ideas are already appearing in the LL programming literature, *e.g.* [Harland *et al.*, 1995, p13] say:

> We would require a LYGON version of Negation-as-Failure, in that we wish to be able to test whether a given resource has been exhausted or not. There is clearly some technical work to be done here, but such a facility, which seems a natural feature for a logic programming language, would significantly enhance the resource-oriented facilities of LYGON.

The LYGON documentation mentions that `once` "will be removed form the language in a future version".

5.3 Conclusions

Our presentation of EC as a LL theory demonstrates the possibility of representing change without involving the heavy machinery of non-monotonic reasoning. One advantage of using LL in this way over most non-monotonic logics is that they depend on the notion of non-derivability ("if P is not derivable then derive Q"), whereas change in LL is a straightforward deductive operation. The main advantages of LL over modal logics and their descendants (temporal logics, dynamic logic) are:

Locality LL change is local to the involved property/predicate, whereas modal logic has to go to a completely new world which is only indirectly related (through the accessibility relation) to the current world.

Simplicity LL was conceived on the road to simplicity: certain structural rules of classical logic were banned. Modal logic, on the other hand, was conceived by enriching the set of connectives, and by making semantical structures much more complex.

Of course, "full-strength" NMR is indispensable in certain reasoning contexts, for example common-sense reasoning. We argue however that it has been overused in contexts where it is better left alone.

References

[Alexiev, 1994] V. Alexiev. Applications of linear logic to computation: An overview. *Bulletin of the IGPL*, 2(1):77–107, March 1994. Also University of Alberta TR93–18, December 1993.

[Arima and Sawamura, 1993] J. Arima and H. Sawamura. Reformulation of explanation by linear logic: Toward logic for explanation. In K. P Jantke, editor, *4th International workshop on Algorithmic learning theory*, number 744 in LNCS, pages 45–58, Tokyo, Japan, November 1993.

[Cervesato *et al.*, 1994] I. Cervesato, L. Chittaro, and A. Montanari. Modal event calculus in LOLLI. Technical Report CMU-CS-94-198, Carnegie Mellon University, Pittsburgh, PA, September 1994.

[Fouqueré and Vauzeilles, 1994] C. Fouqueré and J. Vauzeilles. Linear logic and exceptions. *Journal of Logic and Computation*, 4(6):859–875, 1994.

[Gelfond and Lifschitz, 1993] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.

[Girard, 1987] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Girard, 1993] J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[Hanks and McDermott, 1987] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[Harel, 1979] D. Harel. *First-Order Dynamic Logic*, volume 68 of *LNCS*. Springer-Verlag, 1979.

[Harland and Pym, 1992] J. Harland and D. Pym. On resolution in fragments of classical linear logic (extended abstract). In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LPAR'92)*, number 624 in LNAI (subseries of LNCS), pages 30–41, St. Petersburg, Russia, July 1992.

[Harland and Pym, 1994] J. Harland and D. Pym. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, April 1994.

[Harland and Winikoff, 1995] J. Harland and M. Winikoff. Implementation and development issues for the linear logic programming language LYGON. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 563–572, Adelaide, Australia, February 1995. Also available as Technical Report TR 95/6, Melbourne University, Department of Computer Science.

[Harland *et al.*, 1995] J. Harland, D. Pym, and M. Winikoff. Programming in LYGON: An overview, April 1995.

[Hodas and Miller, 1994] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 110(2):327–365, May 1994. An extended abstract appeared in LICS'91:32–42, July 1991.

[Hodas, 1992] J. S. Hodas. LOLLI: An extension of λ-PROLOG with linear logic context management. In *1992 λProlog Workshop*, 1992. Available from `ftp.cis.upenn.edu: pub/Lolli`.

[Hölldobler, 1992] S. Hölldobler. On deductive planning and the frame problem. In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LPAR'92)*, number 624 in LNAI (subseries of LNCS), pages 13–29, St. Petersburg, Russia, July 1992.

[Jacopin, 1993] E. Jacopin. Classical AI planning as theorem proving: The case of a fragment of linear logic. In *AAAI Fall Symposium on Automated Deduction in Nonstandard Logics*, pages 62–66, Palo Alto, California, 1993. AAAI Press Publications.

[Kowalski and Sergot, 1986] R. Kowalski and M. Sergot. A logic-based caclulus of events. *New Generation Computing*, (4):67–95, 1986.

[Kowalski, 1992] R. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, pages 121–146, December 1992.

[Masseron *et al.*, 1993] M. Masseron, C. Tollu, and J. Vauzeilles. Generating plans in linear logic: I Actions as proofs. *Theoretical Computer Science*, 113, June 1993. Also in Proc. 10-th Conf. *Foundations of Software Technology and Theoretical Computer Science (FST-"Theoretical Computer Science"'90)*, Banglore, India, LNCS 472, 1990.

[McCarthy and Hayes, 1969] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. Also appears in N. Nilsson and B. Webber (editors), Readings in Artificial Intelligence, Morgan-Kaufmann.

[Pinto and Reiter, 1993] Javier Pinto and Raymond Reiter. Temporal reasoning in logic programming: A case for the situation calculus. In *Intl. Conf. on Logic Programming (ICLP'93)*, pages 203–221, 1993.

[Provetti, 1994] A. Provetti. Hypothetical reasoning: From situation calculus to event calculus. In *Workshop TIME'94*, Pensacola Beach, FL, May 1994. Submitted to Computational Intelligence Journal.

[Sadri and Kowalski, 1995] Fariba Sadri and Robert A. Kowalski. Variants of the event calculus. In *Intl. Conf. on Logic Programming (ICLP'95)*, Tokyo, Japan, June 1995.

[Winikoff and Harland, 1994] M. Winikoff and J. Harland. Implementing the linear logic programming language LYGON. Technical Report 94/23, University of Melbourne, 1994.

This article was processed using the LaTeX macro package with LLNCS style