Learning Effect Axioms via Probabilistic Logic Programming

Rolf Schwitter

Department of Computing, Macquarie University, Sydney NSW 2109, Australia
`Rolf.Schwitter@mq.edu.au`

Abstract

Events have effects on properties of the world; they initiate or terminate these properties at a given point in time. Reasoning about events and their effects comes naturally to us and appears to be simple, but it is actually quite difficult for a machine to work out the relationships between events and their effects. Traditionally, effect axioms are assumed to be given for a particular domain and are then used for event recognition. We show how we can automatically learn the structure of effect axioms from example interpretations in the form of short dialogue sequences and use the resulting axioms in a probabilistic version of the Event Calculus for query answering. Our approach is novel, since it can deal with uncertainty in the recognition of events as well as with uncertainty in the relationship between events and their effects. The suggested probabilistic Event Calculus dialect directly subsumes the logic-based dialect and can be used for exact as well as a for inexact inference.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Effect Axioms, Event Calculus, Event Recognition, Probabilistic Logic Programming, Reasoning under Uncertainty

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.8

1 Introduction

The Event Calculus [9] is a logic language for representing events and their effects and provides a logical foundation for a number of reasoning tasks [13, 24]. Over the years, different versions of the Event Calculus have been successfully used in various application domains; amongst them for temporal database updates, for robot perception and for natural language understanding [12, 13]. However, many event recognition scenarios exhibit a significant amount of uncertainty, since a system may not be able to detect all events reliably and the effects of events may not always be known in advance. In order to deal with uncertainty, we have to extend the logic-based Event Calculus with probabilistic reasoning capabilities and try to learn the effects of events from example interpretations. Effect axioms are important in the context of the Event Calculus, since they specify which properties are initiated or terminated when a particular event occurs at a given point in time.

Recently, the combination of logic programming and probability under the distribution semantics [5, 23] has proven to be useful for building rich representations of domains consisting of individuals and uncertain relations between these individuals. These representations can be learned in an efficient way and used to carry out inference. The distribution semantics underlies a number of probabilistic logic languages such as PRISM [23], Independent Choice Logic [14, 15], Logic Programs with Annotated Disjunctions [27], P-log [1], and ProbLog [4, 6]. Since these languages have the same formal foundation, there exist linear transformations between them that preserve their semantics [20].

In this paper, we investigate how the language of Logic Programs with Annotated Disjunctions (LPAD) and its implementation in the *cplint* framework of programs for reasoning with probabilistic logic programs [22] can be used to learn the structure of probabilistic effect axioms for a dialect of the Event Calculus.

Recently, Skarlatidis and colleagues introduced two probabilistic dialects of the Event Calculus for event recognition, in particular for the detection of short-term activities in video frames. Their first dialect, Prob-EC [25], is based on probabilistic logic programming [8] and handles noise in the input data. Input events are assumed to be independent and are associated with detection probabilities. Their second dialect, MLN-EC [26], is based on Markov logic networks [16] and does not make any independence assumption of input events. Our probabilistic dialect of the Event Calculus is related to Prob-EC in the sense that it is based on the distribution semantics. However, we focus in our work not only on the processing of uncertain events but also on the learning of the structure and parameters of effect axioms from example interpretations, an issue that has not been addressed by Skarlatidis and colleagues.

The rest of this paper is structured as follows: In Section 2, we introduce our dialect of the logic-based Event Calculus, followed by a brief introduction to probabilistic logic programming in Section 3. In Section 4, we reformulate the logic-based Event Calculus as a probabilistic logic program where the events and the effect axioms are annotated with probabilities. In Section 5, we discuss how we can learn the structure of these effect axioms from positive and negative interpretations that are available in the form of short dialogue sequences. In Section 6, we present our experiments and show that the proposed probabilistic dialect is an elaboration-tolerant version of the logic-based Event Calculus. In Section 7, we summarise the advantages of our approach and present our conclusion.

2 The Event Calculus

The original logic-based Event Calculus as introduced by [9] is a logic programming formalism for representing the effects of events on properties. The basic ontology of the Event Calculus consists of events, fluents, and time points. An *event* represents an action that may occur in the world; for example, a person who is arriving in the kitchen. A *fluent* represents a time-varying property that might be the effect of an event; for example, a person who is located in the kitchen (after arriving in the kitchen). A *time point* represents an instant of time and indicates when an event happens or when a fluent holds; for example, Sunday, 19-Feb-17, 23:59:00, UTC-12[1]. In the following discussion we introduce the axioms of our dialect of the Simple Event Calculus (SEC) [24]. These axioms are implemented as Prolog clauses and displayed in Listing 1.

Listing 1 The Simple Event Calculus (SEC)

```
holds_at(fluent:F, tp:T) :-                                   % SEC1
  initially(fluent:F),
  \+ clipped(tp:0, fluent:F, tp:T).

holds_at(fluent:F, tp:T2) :-                                  % SEC2
  initiated_at(fluent:F, tp:T1),
  T1 < T2,
  \+ clipped(tp:T1, fluent:F, tp:T2).
```

[1] In the following we use integers instead of POSIX time to save space in the paper.

```
clipped(tp:T1, fluent:F, tp:T3) :-                        % SEC3
  terminated_at(fluent:F, tp:T2),
  T1 < T2, T2 < T3.

initiated_at([fluent:located, pers:A, loc:B], tp:C) :-    % EAX1
  happens_at([event:arrive, pers:A, loc:B], tp:C).

terminated_at([fluent:located, pers:A, loc:D], tp:C) :-   % EAX2
  happens_at([event:arrive, pers:A, loc:B], tp:C),
  B \= D.

initially([fluent:located, pers:bob, loc:garden]).        % SCO1

happens_at([event:arrive, pers:bob, loc:kitchen], tp:3).  % SCO2

happens_at([event:arrive, pers:bob, loc:garage], tp:5).   % SCO3
```

Axiom `SEC1` specifies that a fluent `F` initially holds at time point `T1`, if it held at time point `0`, and has not been terminated between these two time points. Axiom `SEC2` specifies that a fluent `F` holds at time point `T2`, if the fluent has been initiated at some time point `T1`, which is before `T2` and has not been clipped between `T1` and `T2`. Axiom `SEC3` states that a fluent `F` has been clipped between time point `T1` and `T3`, if the fluent has been terminated at a time point `T2` and this time point is between `T1` and `T3`. Note that according to these domain-independent axioms (`SEC1-SEC3`), a fluent does not hold at the time of the event that initiates it but at the time of the event that terminates it.

Events have effects on properties of the world; they initiate and terminate these properties at a given point in time. These effects can be described by domain-dependent effect axioms. For example, the positive effect axiom `EAX1` in Listing 1 specifies that the fluent with the name `located` involving a person `A` and a location `B` is initiated after the time point `C`, if an event occurs at a time point `C` where the person `A` arrives at the location `B`. The negative effect axiom `EAX2` in Listing 1 specifies that the fluent with the name `located` involving a person `A` and a location `D` is terminated after the time point `C`, if an event occurs at a time point `C` where the person `A` arrives at a location `B` that is different from location `D`. Finally, a scenario (`SCO1-SCO3`) is required where the initial state of the world (`SCO1`) is described as well as a sequence of events (`SCO2` and `SCO3`) that occur at subsequent time points.

This setting allows us to investigate which fluents hold at a given point in time. The logic-based SEC assumes that the effect axioms are known in advance and that there is no uncertainty in the relationships between events and effects and in the recognition of events. In the following, we assume that the effect axioms are unknown and need to be learned first from example interpretations and that the recognition of events that occur in the real world can be noisy. During the learning process, the structure of the effect axioms will be generated automatically and the resulting axioms will be annotated with probabilities. These probabilistic effect axioms can then be processed with a version of the SEC that is based on a probabilistic logic programming language which supports probabilistic reasoning.

3 Probabilistic Logic Programs (PLP)

One of most successful approaches to Probabilistic Logic Programs (PLP) is based on the distribution semantics [23]. Under the distribution semantics a probabilistic logic program defines a probability distribution over a set of normal logic programs (called *worlds*). The probability of a query is then obtained from the joint distribution of the query and the worlds by marginalization.

While the distribution semantics underlies a number of different probabilistic languages (see [20] for an overview), Logic Programs with Annotated Disjunctions (LPAD) [27] offer the most general syntax of this family of languages. An LPAD P consists of a set of annotated disjunctive clauses C_i of the form:

$h_1 : \alpha_1; \ldots ; h_n : \alpha_n \leftarrow b_1, \ldots , b_m.$

where h_i are logical atoms, α_i are real numbers, each of them standing for a probability in the interval $[0, 1]$ such that the sum of all α_i is 1, and b_i are logical literals (incl. negation as failure). The set of elements $h_i : \alpha_i$ form the head of a clause and the set of elements b_i the body. Disjunction in the head is represented by a semicolon and the atoms in the head a separated by a colon from their probabilities. Note that if $n = 1$ and $\alpha_1 = 1$, then a clause corresponds to a normal clause and the annotation can be omitted. Note also that if the sum of all α_i is smaller than 1, then an additional disjunct *null* is assumed with probability $1 - sum(\alpha_i)$. If the body of a clause is empty, then it can be omitted.

The semantics of an LPAD P is defined via its grounding. The grounding of P is obtained by replacing the variables of each clause C with the terms of the Herbrand universe $H_U(P)$ [10]. Each of these ground clauses represents a probabilistic choice between a number of non-disjunctive clauses. By selecting a head atom for each ground clause of an LPAD, we get an instance of a normal logic program. Each selection has its own probability assigned to it and the product of these probabilities induces the probability of a program instance, assuming independence among the choices made for each clause. All instances of an LPAD together define a probability distribution over a set of interpretations of the program. The probability of a particular interpretation I is then the sum of the probability of all instances for which I is a model (see [27] for details).

4 The Simple Event Calculus as a PLP

We can reformulate the logic-based SEC as a PLP and process it with the help of the PITA library [18] that runs in SWI Prolog[2]. PITA computes the probability of a query from an LPAD program by transforming the program into a normal program that contains calls to manipulate Binary Decision Diagrams as auxiliary data structures and is evaluated by Prolog. PITA was compared with ProbLog [4] and found to be fast and scalable [17]. Alternatively, we can use MCINTYRE [19], if exact inference in PITA gets too expensive. MCINTYRE performs approximate inference using Monte Carlo sampling. Both PITA and MCINTYRE are part of the *cplint* framework[3] for reasoning with probabilistic logic programs.

In contrast to the logic-based dialect of the SEC, we have to load the `pita` (or `mcintyre`) library first, initialise it with a corresponding directive and enclose the LPAD clauses for the SEC in additional directives that mark the start and end of the program as illustrated in Listing 2:

Listing 2 PITA Initialisation

```
:- use_module(library(pita)).
:- pita.
```

[2] http://www.swi-prolog.org
[3] https://github.com/friguzzi/cplint

```
:- begin_lpad.

   % The SEC with probabilistic events and probabilistic effect
   % axioms goes here.

:- end_lpad.
```

The domain-independent clauses for the probabilistic version of the SEC are the same (`SEC1-SEC3`) as those of the logic-based version introduced in Listing 1. The probabilistic effect axioms that we are going to learn (see Section 5) have the same basic form as the axioms `EAX1` and `EAX2`, but they are additionally annotated with probabilities and contain special conditions in the body of the clauses to guarantee that all variables in the head of a clause are range restricted; otherwise the distribution semantics is not well-defined for an LPAD program. In our case, the annotated effect axioms look as illustrated in Listing 3:

Listing 3 Effect Axioms with Probabilities

```
initiated_at([fluent:located, pers:A, loc:B], tp:C):0.66;'':0.34 :-
  happens_at([event:arrive, pers:A, loc:B], tp:C).

terminated_at([fluent:located, pers:A, loc:D], tp:C):0.66;'':0.34 :-
  happens_at([event:arrive, pers:A, loc:B], tp:C),
  location([loc:D]).
```

Here, the value `0.66` stands for the probability of the clause to be true and the value `0.34` for the probability of the clause to be false. Note that the predicate `location/1` in the body of the second clause restricts the range of the variable `D` in the head of the clause. That means we have to make sure – as we will see in Section 5.2 – that facts for the locations are available in the background knowledge for the LPAD program that is used to learn the structure of these axioms.

In order to deal with uncertainty in the initial setting and the recognition of events, we can also add probabilities to the facts that describe the scenario (`SCO1-SCO3`); for example, to the facts `SCO2` and `SCO3` that describe the events as shown in Listing 4:

Listing 4 Events with Probabilities

```
happens_at([event:arrive, pers:bob, loc:kitchen], tp:3):0.95;'':0.05.
happens_at([event:arrive, pers:bob, loc:garage],  tp:5):0.99;'':0.01.
```

If the annotation of the probability is omitted in a clause, then its value is implicitly `1`. As we will see in Section 6, the probabilistic version of the SEC without annotated probabilities behaves exactly like the logic-based version of the SEC. In this respect, the probabilistic version of the SEC can be considered as an elaboration-tolerant extension of the logic-based version, since all modifications that are required for building the resulting probabilistic logic program are additive.

5 Learning the Structure of Effect Axioms

To learn the structure of effect axioms from positive and negative interpretations, we use a separate LPAD program together with SLIPCOVER [3], an algorithm for learning the structure and parameters of probabilistic logic programs. SLIPCOVER takes as input a set of example interpretations and a language bias that indicates which predicates are target. These interpretations must contain positive and negative examples for all predicates that may appear in the head of a clause. SLIPCOVER learns the structure of effect axioms by first performing a beam search in the space of probabilistic clauses and then a greedy search in the space of theories. The first search step starts from a set of bottom clauses, aims at finding a set of promising clauses, and looks for good refinements of these clauses in terms of the log-likelihood of the data. The second search step starts with an empty theory and tries to add each clause for a target predicate to that theory. After each addition, the log-likelihood of the data is computed as the score of the new theory. If the value of the new theory is better than the value of the previous theory, then the clause is kept in the theory, otherwise the clause is discarded. Finally, SLIPCOVER completes a theory consisting of target predicates by adding the body predicates to the clauses and performs parameter learning on the resulting theory (for details see [3]).

Note that the refinements during this process are scored by estimating the log-likelihood of the data by running a small number of iterations of EMBLEM [2], an implementation of expectation-maximization for learning parameters that computes expectations directly on Binary Decision Diagrams.

Listing 5 LPAD Program for Learning the Structure of Effect Axioms with SLIPCOVER

```
:- use_module(library(slipcover)).                                  % 1.1
:- sc.                                                              % 1.2
:- set_sc(max_var, 4).                                              % 1.3
:- set_sc(megaex_bottom, 3).                                        % 1.4
:- set_sc(depth_bound, false).                                      % 1.5
:- set_sc(neg_ex, given).                                           % 1.6

:- begin_bg.                                                        % 2.1
   location([loc: bathroom]).                                       % 2.2
   location([loc: kitchen]).                                        % 2.3
   location([loc: living_room]).                                    % 2.4
:- end_bg.                                                          % 2.5

output(initiated_at/2).                                             % 3.1
input(happens_at/2).                                                % 3.2
modeh(*, initiated_at([fluent: -#fl, pers: +pers, loc: +loc],       % 3.3
                      tp: +tp)).
modeb(*, happens_at([event: -#ev, pers: +pers, loc: +loc],          % 3.4
                    tp: +tp)).
determination(initiated_at/2, happens_at/2).                        % 3.5

initiated_at(ID, [fluent:F2, pers:P, loc:L2], tp:T2) :-             % 4.1
   holds_at(ID, [fluent:_F1, pers:P, loc:_L1], tp:T1),
   happens_at(ID, [event:_E, pers:P, loc:L2], tp:T2),
   T1 < T2,
   holds_at(ID, [fluent:F2, pers:P, loc:L2], tp:T3),
   T2 < T3.

neg(initiated_at(ID, [fluent:F2, pers:P, loc:_L], tp:T2)) :-        % 4.2
   holds_at(ID, [fluent:_F1, pers:P, loc:_L1], tp:T1),
   happens_at(ID, [event:_E, pers:P, loc:L2], tp:T2),
   T1 < T2,
   neg(holds_at(ID, [fluent:F2, pers:P, loc:L2], tp:T3)),
   T2 < T3.
```

```
begin(model(f1)).                                              % 5.1
   holds_at([fluent:located, pers:mary, loc:bathroom], tp:0).
   happens_at([event:arrive, pers:mary, loc:kitchen], tp:1).
   holds_at([fluent:located, pers:mary, loc:kitchen], tp:2).
end(model(f1)).

begin(model(f2)).                                              % 5.2
   holds_at([fluent:located, pers:emma, loc:living_room], tp:3).
   happens_at([event:arrive, pers:emma, loc:bathroom], tp:4).
   neg(holds_at([fluent:located, pers:emma, loc:bathroom], tp:5)).
end(model(f2)).

begin(model(f3)).                                              % 5.3
   holds_at([fluent:located, pers:sue, loc:kitchen], tp:6).
   happens_at([event:arrive, pers:sue, loc:living_room], tp:7).
   holds_at([fluent:located, pers:sue, loc:living_room], tp:8).
end(model(f3)).

fold(train, [f1, f2, f3]).                                     % 5.4

learn_effect_axioms(C) :-                                      % 5.5
  induce([train], C).
```

In our case, the LPAD program for structure learning consists of five parts: (1) a preamble, (2) background knowledge for type definitions, (3) language bias information, (4) clauses for finding examples, and (5) example interpretations (= models). In the following subsections, we discuss these parts in more detail.

5.1 Preamble

In the preamble of the program, the SLIPCOVER library is loaded (1.1) and initialised (1.2), and the relevant parameters are set (1.3-1.6). In our case, these parameters are: the maximum number of distinct variables that can occur in a clause to be learned (1.3); the number of examples on which to build the bottom clauses (1.4); the depth of the derivation which is unbound in our case, and therefore `false` (1.5); and the availability of negative examples in the interpretations (1.6).

5.2 Background Knowledge

The background knowledge specifies the kind of knowledge that is valid for all interpretations of a clause. This knowledge is enclosed in a begin directive (2.1) and an end directive (2.5). In our case, it contains type definitions for different locations (2.2-2.4) that are required to restrict the range of variables in the clauses to be learned. These are predicates that may be used for constructing the body of a clause. As we will see in Section 5.5, these type definitions are automatically derived from the same dialogue sequences that are used to construct the example interpretations.

5.3 Language Bias Information

The language bias specifies the accepted structure of the clauses to be learned and helps to guide the construction of the refinements for the resulting theory. The language bias is expressed in terms of: (a) predicate declarations, (b) mode declarations, (c) type specifications, and (d) determination statements.

(a) **Predicate declarations** take the form of output predicates (3.1) or input predicates (3.2). Output predicates are declared as `output/1` and specify those predicates whose atoms one intends to predict. Input predicates are declared as `input/1` and specify those predicates whose atoms one is not interested in predicting but which should occur in the body of a hypothesized clause.

(b) **Mode declarations** are used to guide the process of constructing a generalisation from example interpretations and to constrain the search space for the resulting clauses. We distinguish between head mode declarations (3.3) and body mode declarations (3.4). A head mode declaration (`modeh(n, atom)`) specifies the atoms that can occur in the head of a clause and a body mode declaration (`modeb(n, atom)`) those atoms that can occur in the body of a clause. The argument `n`, the recall, is either an integer ($n \geq 1$) or an asterisk (`*`) and indicates how many atoms for the predicate specification are retained in the bottom clause during a saturation step. The asterisk stands for all those atoms that are found; otherwise the indicated number of atoms is randomly chosen.

(c) **Type specifications** have, in our case, the form `+type`, `-type`, or `-#type`, and specify that the argument should be either an input variable (`+`) of that type, an output variable (`-`) of that type, or a constant (`-#`) of that type. For example, the argument of the form `-#fl` in the head mode declaration of (3.3) stands for the name of a fluent, and the argument of the form `-#ev` in the body mode declaration of (3.4) stands for the name of an event, and symbols prefixed with `+` and `-` for input and output variables.

(d) **Determination statements** such as (3.5) are required by SLIPCOVER and declare which predicates can occur in the body of a particular clause.

In addition to the specification of the language bias for the positive effect axiom (`initiated_at/2`) in Listing 5, we present below in Listing 6 the specification for the negative effect axiom (`terminated_at/2`), since the successful construction of this negative effect axiom depends on some important additions:

Listing 6 Negative Effect Axiom

```
output(terminated_at/2).
input(happens_at/2)

modeh(*, terminated_at([fluent: -#fl, pers: +pers, loc: +loc2],
                        tp: +tp)).
modeb(*, happens_at([event: -#ev, pers: +pers, loc: -loc1], tp: +tp)).
modeb(*, location([loc: +loc2])).

determination(terminated_at/2, happens_at/2).
determination(terminated_at/2, location/1).

lookahead_cons_var(location([loc:_L2]),
                     [happens_at([event:_E, pers:_P, loc:_L1], tp:_T)]).
lookahead_cons_var(happens_at([event:_E, pers:_P, loc:_L1], tp:_T),
                     [location([loc:_L2])]).
```

Here, the two determination statements indicate that the predicate `happens_at/2` as well as the predicate `location/1` can appear in the body of the negative effect axiom. Additionally, we have to specify a lookahead that enforces that whenever one of these two predicates is added to the body of the clause during refinement, then also the other predicate needs to be added to that body.

5.4 Program Clauses for Finding Examples

The program clauses for finding examples (4.1 + 4.2) in Listing 5 contain those predicates that are used during search. These clauses are not part of the background knowledge and therefore contain an additional argument (`ID`) that is used to identify the relevant example interpretations (models). We encode the search for finding examples intensionally as the clauses in Listing 5 illustrate. This representation completes the interpretations by generating positive and negative examples for the positive effect axiom (`initiated_at/2`) using the predefined predicate `neg/1` in the clause (4.2). To complete our discussion, we show below in Listing 7 the clauses for finding examples for the negative effect axiom (`terminated_at/2`):

Listing 7 Finding Negative Effect Axiom

```
terminated_at(ID, [fluent:F1, pers:P, loc:L1], tp:T2) :-
  holds_at(ID, [fluent:F1, pers:P, loc:L1], tp:T1),
  happens_at(ID, [event:_E, pers:P, loc:L2], tp:T2),
  T1 < T2,
  holds_at(ID, [fluent:_F2, pers:P, loc:L2], tp:T3),
  T2 < T3.

neg(terminated_at(ID, [fluent:F1, pers:P, loc:L1], tp:T2)) :-
  holds_at(ID, [fluent:F1, pers:P, loc:L1], tp:T1),
  happens_at(ID, [event:_E, pers:P, loc:L2], tp:T2),
  T1 < T2,
  neg(holds_at(ID, [fluent:_F2, pers:P, loc:L2], tp:T3)),
  T2 < T3.
```

The interesting thing to note here is that the variable `L1` for the location becomes available via the example interpretation, but the predicate `location/1` that is used to restrict the range of this variable comes from the background information and is enforced via the lookahead predicate discussed in the previous section.

5.5 Example Interpretations

In our case the example interpretations (5.1-5.3) in Listing 5 are derived from short dialogue sequences. We experimented with dialogue sequences consisting of a state sentence, followed by an event sentence, followed by a question that results in a positive or negative answer. These dialogue sequences are similar to the data used in the dialogue-based language learning dataset [28], and look in our case as follows:

S.1.1 Mary is located in the bathroom.
S.1.2 Mary arrives in the kitchen.
S.1.3 Where is Mary?
S.1.4 In the kitchen.

S.2.1 Emma is located in the living room.
S.2.2 Emma arrives in the bathroom.
S.2.3 Is Emma in the bathroom?
S.2.4 No.

These dialogues are parsed into dependency structures and automatically translated into the corresponding Event Calculus representation. For this purpose, we used the Stanford Parser [11] that generates for each sentence (or answer fragment) a dependency structure.

This dependency structure is then translated with the help of a simple ontology into the corresponding Event Calculus notation. The ontology contains, among other things, the information that Mary is a person and that a bathroom is a location. The dialogue sequence S.1.1-S.1.4, for example, is first translated into four dependency structures as illustrated on the left-hand side of Listing 8:

Listing 8 From Dependency Structures to Event Calculus Representation

```
DEPENDENCY STRUCTURES               EVENT CALCULUS REPRESENTATION

% D.1.1
nsubjpass(located-3, Mary-1)        holds_at([fluent:located,
auxpass(located-3, is-2)                      pers:mary,
root(ROOT-0, located-3)                       loc:bathroom], tp:1).
case(bathroom-6, in-4)
det(bathroom-6, the-5)              location([loc:bathroom]).
nmod(located-3, bathroom-6)

% D.1.2
nsubj(arrives-2, Mary-1)            happens_at([event:arrive,
root(ROOT-0, arrives-2)                         pers:mary,
case(kitchen-5, in-3)                           loc:kitchen], tp:2).
det(kitchen-5, the-4)
nmod(arrives-2, kitchen-5)          location([loc:kitchen]).

% D.1.3
advmod(is-2, Where-1)
root(ROOT-0, is-2)
nsubj(is-2, Mary-3)                 holds_at([fluent:located,
                                              pers:mary,
% D.1.4                                       loc:kitchen], tp:3).
case(kitchen-3, In-1)
det(kitchen-3, the-2)
root(ROOT-0, kitchen-3)
```

For example, the dependency structure D.1.1 for the sentence S.1.1 is translated into a `holds_at/2` predicate, since the sentence describes a fluent. The translation of the dependency structure D.1.2 for sentence S.1.2 results in a `happens_at/2` predicate, since the sentence describes an event. Finally, the translation of the two dependency structures D.1.3 + D.1.4 for the question-answer pair in S.1.3 + S.1.4 introduce a positive `holds_at/2` predicate, since the question and the answer together confirm that a particular fluent holds. Note that this dialogue sequence is also used to derive together with the help of the ontology the factual information that bathroom and kitchen are locations.

In our LPAD program for structure learning in Listing 5, the derived example interpretations (5.1-5.3) are initiated by predicates of the form `begin(model(<name>))` and terminated by predicates of the form `end(model(<name>))` and the relevant background information is added to the background section (2.1-2.5) of the program.

Note that each example interpretation may contain an additional fact of the form `prob(P)` that assigns a probability `P` to the interpretation. This probability may be used to reflect the confidence of the parser in a particular interpretation. If this probability is omitted, then the probability of each interpretation is considered equal to `1/n` where `n` is the total number of interpretations (for details see [22]).

Finally, we have to specify how the example interpretations are divided in folds for taining (5.4), before we can perform parameter learning on the training fold as illustrated in (5.5).

6 Experiments

In order to get more realistic probabilities for the effect axioms that we discussed in this paper, we learned the structure of these axioms with the help of 50 example dialogues. This resulted in the following probabilities for the two effect axioms as shown in Listing 9:

Listing 9 Effect Axioms with Probabilities Derived from Example Dialogues

```
initiated_at([fluent:located, pers:A, loc:B], tp:C):0.87;'':0.13 :-
  happens_at([event:arrive, pers:A, loc:B], tp:C).

terminated_at([fluent:located, pers:A, loc:D], tp:C):0.87;'':0.13 :-
  happens_at([event:arrive, pers:A, loc:B], tp:C),
  location([loc:D]).
```

We used the same domain-independent axioms (`SEC1-SEC3`) for our experiments as in the logic-based version of the SEC in Listing 1, but added the following background axioms in Listing 10 to the probabilistic version of the program to deal with the range requirement for variables of the learned clauses under the distribution semantics:

Listing 10 Background Axioms

```
location([loc: garage]).
location([loc: garden]).
location([loc: kitchen]).
```

To test the probabilistic dialect of the SEC, we conducted three experiments using PITA and MCINTYRE for reasoning and the queries shown in Listing 11 + 12. In the first experiment A, we removed all probabilities from the axioms and executed the queries; in the second experiment B, we used the effect axioms annotated with the learned probabilities to answer the queries; and finally in the third experiment C, we used the annotated effect axioms together with probabilities for noisy event occurrences and executed the queries. We then run the same three experiments using MCINTYRE and sampled the answer for each query 100 times. This sampling process returns the estimated probability that a sample is true (i.e., that a sample succeeds).

Listing 11 Test Queries used to Evaluate the SEC with PITA

```
test_ec_pita(Num, [P1, P2, P3, P4, P5, P6, P7]) :-
  prob( holds_at([fluent:located, pers:bob, loc:garden],  tp:1, P1 ),
  prob( holds_at([fluent:located, pers:bob, loc:garden],  tp:2, P2 ),
  prob( holds_at([fluent:located, pers:bob, loc:kitchen], tp:3, P3 ),
  prob( holds_at([fluent:located, pers:bob, loc:kitchen], tp:4, P4 ),
  prob( holds_at([fluent:located, pers:bob, loc:kitchen], tp:5, P5 ),
  prob( holds_at([fluent:located, pers:bob, loc:garage],  tp:5, P6 ),
  prob( holds_at([fluent:located, pers:bob, loc:garage],  tp:6, P7 ).
```

Listing 12 Test Queries used to Evaluate the SEC with MCINTYRE

```
test_ec_mcintyre([P1, P2, P3, P4, P5, P6, P7]) :-
  mc_sample( holds_at([fluent:located, pers:bob, loc:garden],  tp:1),
             100, P1 ),
  mc_sample( holds_at([fluent:located, pers:bob, loc:garden],  tp:2),
             100, P2 ),
  mc_sample( holds_at([fluent:located, pers:bob, loc:kitchen], tp:3),
             100, P3 ),
  mc_sample( holds_at([fluent:located, pers:bob, loc:kitchen], tp:4),
             100, P4 ),
  mc_sample( holds_at([fluent:located, pers:bob, loc:kitchen], tp:5),
             100, P5 ),
  mc_sample( holds_at([fluent:located, pers:bob, loc:garage],  tp:5),
             100, P6 ),
  mc_sample( holds_at([fluent:located, pers:bob, loc:garage],  tp:6),
             100, P7 ).
```

6.1 Experiment A

For the first experiment, we removed the probabilities of the learned effect axioms and automatically combined them with the domain-independent axioms (`SEC1-SEC3`) and the axioms of the original scenario (`SCO1-SCO3`), and then executed the queries in Listing 11 and 12 using the two inference modules PITA (`P`) and MCINTYRE (`M`). This resulted in the following answers:

```
(P) Probs = [1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0]
(M) Probs = [1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0]
```

Here the value `1.0` stands for "true" and the value `0.0` for "false" and the answers are the same as we get for the logic-based version of the SEC. Note that by definition a fluent does not hold at the time point of the event that initiates it; therefore, the probability for the answer of the third query and the sixth query in Listing 11 + 12 is `0.0`.

6.2 Experiment B

In this experiment, we used the probabilistic effect axioms that we learned with the help of our 50 training examples. This gives the following results for our test queries:

```
(P) Probs = [1.0, 1.0, 0.0, 0.87, 0.87, 0.0, 0.87]
(M) Probs = [1.0, 1.0, 0.0, 0.85, 0.91, 0.0, 0.87]
```

Note that the first three answers and the sixth answer are the same as before, answer four, five and seven show the probabilistic effect axioms at work. Since MCINTYRE uses inexact inference and relies on sampling, the results for each run show some variation. We observe a standard deviation of about 0.034 for the relevant answers when we run each query ten times and use 100 samples for each query (see Listing 12).

6.3 Experiment C

For our third experiment, we used again the probabilistic effect axioms and added probabilities to the events to deal with a situation where we have noise in the recognition of events:

```
happens_at([event:arrive,pers:bob,loc:kitchen],tp:3):0.95;'':0.05.
happens_at([event:arrive,pers:bob,loc:garage],tp:5):0.99;'':0.01.
```

Running our test queries under these uncertain conditions gives the following results:

```
(P) Probs = [1.0, 1.0, 0.0, 0.8265, 0.8265, 0.0, 0.8613]
(M) Probs = [1.0, 1.0, 0.0, 0.84, 0.82, 0.0, 0.84]
```

As expected, uncertainty in event recognition lowers the probability for the relevant answers, and there is of course some variation under inexact inference for each run.

7 Conclusion

In this paper we showed how we can automatically learn the structure and parameters of probabilistic effect axioms for the Simple Event Calculus (SEC) from positive and negative example interpretations stated as short dialogue sequences in natural language. We used the *cplint* framework for this task that provides libraries for structure and parameter learning and for answering queries with exact and inexact inference. The example dialogues that are used for learning the structure of the probabilistic logic program are parsed into dependency structures and then further translated into the Event Calculus notation with the help of a simple ontology. The novelty of our approach is that we can not only process uncertainty in event recognition but also learn the structure of effect axioms and combine these two sources of uncertainty to successfully answer queries under this probabilistic setting. Interestingly, our extension of the logic-based version of the SEC is completely elaboration-tolerant in the sense that the probabilistic version fully includes the logic-based version. This makes it possible to use the probabilistic version of the SEC in the traditional way as well as when we have to deal with uncertainty in the observed world. In the future, we would like to extend the probabilistic version of the SEC to deal – among other things – with concurrent actions and continuous change.

Acknowledgements. I would like to thank Fabrizio Riguzzi for his valuable help with the implementation of the algorithm for learning the effect axioms.

References

1 Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. In *Theory and Practice of Logic Programming*, 9(1), pp. 57–144, 2009.

2 Elena Bellodi and Fabrizio Riguzzi. Expectation Maximization over binary decision diagrams for probabilistic logic programs. In *Intelligent Data Analysis*, 17(2), pp. 343–363, 2013.

3 Elena Bellodi and Fabrizio Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. In *Theory and Practice of Logic Programming*, 15(2), pp. 169–212, 2015.

4 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, Hyderabad, India, pp. 2462–2467, 2007.

5 Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. In *Machine Learning*, Vol. 100, Issue 1, Springer New York LLC, pp. 5–47, 2015.

6 Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer and Luc De Raedt. ProbLog2: Probabilistic logic programming. In *Machine Learning and Knowledge Discovery in Databases*, LNCS 9286, Springer, pp. 312–315, 2015.

7 Nikos Katzouris, Alexander Artikis, and Georgios Paliouras. Incremental learning of event definitions with Inductive Logic Programming. In *Machine Learning*, Vol. 100, Issue 2, pp. 555–585, 2015.

8 Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. In: *Theory and Practice of Logic Programming*, Vol. 11, pp. 235–262, 2011.

9 Robert Kowalski and Marek Sergot. A Logic-Based Calculus of Events. In *New Generation Computing*, Vol. 4, pp. 67–95, 1986.

10 John W. Lloyd. *Foundations of logic programming.* Second, Extended Edition. Springer-Verlag, New York, 1987.

11 Christopher, D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60, 2014.

12 Rob Miller and Murray Shanahan. Some Alternative Formulations of the Event Calculus. In *Computational Logic: Logic Programming and Beyond – Essays in Honour of Robert A. Kowaski*, LNAI 2408, Springer pp. 452–490, 2002.

13 Erik T. Mueller. *Commonsense Reasoning, An Event Calculus Based Approach.* 2nd Edition, Morgan Kaufmann/Elesevier, 2015.

14 David Poole. The independent choice logic for modelling multiple agents under uncertainty. In *Artificial Intelligence* Vol. 94, pp. 7–56, 1997.

15 David Poole. The independent choice logic and beyond. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton (eds.), *Probabilistic Inductive Logic Programming: Theory and Application*, LNAI Vol. 4911, Springer, pp. 222–243, 2008.

16 Matthew Richardson and Pedro Domingos. Markov logic networks. In *Machine Learning*, Vol. 62, Issue 1, pp. 107–136, 2006.

17 Fabrizio Riguzzi and Terrance Swift. An extended semantics for logic programs with annotated disjunctions and its efficient implementation. In *Italian Conference on Computational Logic.* CEUR Workshop Proceedings, Vol. 598. Sun SITE Central Europe, 2010.

18 Fabrizio Riguzzi and Terrance Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. In *Theory and Practice of Logic Programming, 27th International Conference on Logic Programming (ICLP'11) Special Issue*, 11(4-5), pp. 433–449, 2011.

19 Fabrizio Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. In: *Fundamenta Informaticae*, 124(4), pp. 521–541, 2013.

20 Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese. A History of Probabilistic Inductive Logic Programming. In *Frontiers in Robotics and AI*, 18. September 2014.

21 Fabrizio Riguzzi and Terrance Swift. Probabilistic logic programming under the distribution semantics. In M. Kifer and Y. A. Liu, (eds), *Declarative Logic Programming: Theory, Systems, and Applications*, LNCS. Springer, 2016.

22 Fabrizio Riguzzi. cplint Manual. SWI-Prolog Version. July 4, 2017.

23 Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In L. Stearling (ed.), *12th International Conference on Logic Programming*, Cambridge: MIT Press, pp. 715–729, 1995.

24 Murray Shanahan. The Event Calculus Explained. In M.J. Wooldridge and M. Veloso (eds), *Artificial Intelligence Today*, LNAI, Vol. 1600, Springer, pp. 409–430, 1999.

25 Anastasios Skarlatidis, Alexander Artikis, Jason Filippou, Georgios Paliouras. A Probabilistic Logic Programming Event Calculus. In *Theory and Practice of Logic Programming*, Vol. 15, No. 2, pp. 213–245, 2015.

26 Anastasios Skarlatidis, Georgios Paliouras, Alexander Artikis, and George A. Vouros. Probabilistic Event Calculus for Event Recognition. In *ACM Transactions on Computational Logic*, Vol. 16, No. 2, Article 11, 2015.

27 Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming* LNCS 3131, Berlin: Springer, pp. 195–209, 2004.

28 Jason Weston. Dialog-based Language Learning. Facebook AI Research. In *arXiv:1604.06045v7*, 24th October 2016.