Logic-Based Representation, Reasoning and Machine Learning for Event Recognition

Alexander Artikis[1], Georgios Paliouras[1], François Portet[2] and Anastasios Skarlatidis[1 3]
[1]Institute of Informatics & Telecommunications, NCSR "Demokritos", Athens 15310, Greece
[2]Laboratoire D'informatique de Grenoble, Grenoble Universités, 38400 Saint Martin d'Hères, France
[3]Department of Information & Communication Systems Engineering, University of the Aegean, Greece
{a.artikis, paliourg, anskarl}@iit.demokritos.gr, Francois.Portet@imag.fr

ABSTRACT

Today's organisations require techniques for automated transformation of the large data volumes they collect during their operations into operational knowledge. This requirement may be addressed by employing event recognition systems that detect activities/events of special significance within an organisation, given streams of 'low-level' information that is very difficult to be utilised by humans. Numerous event recognition systems have been proposed in the literature. Recognition systems with a logic-based representation of event structures, in particular, have been attracting considerable attention because, among others, they exhibit a formal, declarative semantics, they haven proven to be efficient and scalable, and they are supported by machine learning tools automating the construction and refinement of event structures. In this paper we review representative approaches of logic-based event recognition, and discuss open research issues of this field.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

General Terms

Languages

Keywords

event recognition

1. INTRODUCTION

Today's organisations collect data in various structured and unstructured digital formats, but they cannot fully utilise these data to support their resource management. It is evident that the analysis and interpretation of the collected data needs to be automated, in order for large data volumes to be transformed into operational knowledge. Events are particularly important pieces of knowledge, as they represent activities of special significance within an organisation. Therefore, the *recognition of events* is of utmost importance.

Systems for symbolic event recognition ('event pattern matching', in the terminology of [22]) accept as input a stream of time-stamped low-level events (LLE), which are used to recognise high-level events (HLE) of interest. Consider, for example, the recognition of attacks on nodes of a computer network given the TCP/IP messages, and the recognition of suspicious trader behaviour given the transactions in a financial market. Numerous recognition systems have been proposed in the literature — [37, 17, 7, 22] are but a few examples. Recognition systems with a logic-based representation of event structures, in particular, have been attracting considerable attention. In this paper we will present representative approaches of logic-based event recognition.

Logic-based event recognition systems exhibit a formal, declarative semantics, while non-logic-based event recognition systems often exhibit an informal, procedural semantics. As pointed out in [29], informal semantics constitutes a serious limitation for many real-world applications, where validation and traceability of the effects of events are crucial. Moreover, given that a declarative program states *what* is to be computed, not necessarily *how* it is to be computed, declarative semantics can be more easily applied to a variety of settings, not just those that satisfy some low-level operational criteria. A comparison between, and a comprehensive introduction to, logic-based and non-logic-based event processing systems may be found in [29].

Non-logic-based event recognition systems have proven to be, overall, more efficient than logic-based recognition systems, and, thus, most industrial applications employ the former type of system. However, there are logic-based event recognition systems that have also proven to be very efficient and scalable — we will present such a system in this paper.

Furthermore, logic-based event recognition systems can be, and have been, used in combination with existing non-logic-based enterprise event processing infrastructures and middleware. The Prolog-based Prova system [31], for example, has been used in enterprise event processing networks.

HLE 'definitions' in logic-based systems impose temporal and, possibly, atemporal constraints on subevents, that is, LLE or other HLE. An event recognition system, therefore, should allow for, at the very least, temporal representation and reasoning. In this paper we will review a Chronicle Recognition System (CRS) [16, 11, 12], the Event Calculus (EC) [20, 6, 23, 24, 30, 31, 29, 1], and Markov Logic Networks (MLN) [10, 34]. CRS is a purely temporal reasoning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DEBS '10, July 12–15, 2010, Cambridge, UK

Copyright 2010 ACM 978-1-60558-927-5/10/07 ...$10.00.

system that allows for very efficient and scalable event recognition. CRS has been used in various domains, ranging from medical applications to computer network management — see, for example, [12, 4, 3]. EC, which has also been used for event recognition (see, for instance, [5, 30, 31, 29, 1]), allows for the representation of temporal as well as atemporal constraints. Consequently, EC may be used in applications requiring spatial reasoning, for example. Finally, MLN, unlike EC and CRS, allow for uncertainty representation and are thus suitable for event recognition in noisy environments.

The manual development of HLE definitions is a tedious, time-consuming and error-prone process. Moreover, it is often necessary to update HLE definitions during the event recognition process, due to new information about the application under consideration. Consequently, methods for automatically generating and refining HLE definitions from data are highly desirable. For this reason we chose to review approaches that are supported by machine learning techniques. The presentation of each approach, therefore, is structured as follows: representation, reasoning, and machine learning.

To illustrate the reviewed approaches we will use a real-world case study: event recognition for city transport management (CTM). In the context of the PRONTO project[1] an event recognition system is being developed with the aim to support the management of public transport. Buses and trams are equipped with in-vehicle units that send GPS coordinates to a central server offering information about the current status of the transport system (for example, the location of buses and trams on the city map). Additionally, buses and trams are being equipped with sensors for in-vehicle temperature, in-vehicle noise level and acceleration. Given the LLE that will be extracted from these sensors other data sources, such as digital maps, as well as LLE that will be extracted from the communication between the drivers and the public transport control centre, HLE will be recognised related to, among others, the punctuality of a vehicle, passenger and driver comfort, passenger and driver safety, and passenger satisfaction. A detailed description of this case study may be found in [1].

2. A CHRONICLE RECOGNITION SYSTEM

In this section we review the Chronicle Recognition System (CRS) developed by Dousson and colleagues [16, 11, 12].

2.1 Representation

Table 1: Predicates of CRS.

Predicate	Meaning
`event(E, T)`	Event `E` takes place at time `T`
`event(F:(?V1,?V2),T)`	An event takes place at time `T` changing the value of property `F` from `V1` to `V2`
`noevent(E, (T1,T2))`	Event `E` does not take place between `[T1,T2)`
`noevent(F:(?V1,?V2), (T1,T2))`	No event takes place between `[T1,T2)` that changes the value of property `F` from `V1` to `V2`
`hold(F:?V, (T1,T2))`	The value of property `F` is `V` between `[T1,T2)`
`occurs(N,M,E,(T1,T2))`	Event `E` takes place at least `N` times and at most `M` times between `[T1,T2)`

CRS is a temporal reasoning system that has been developed for event recognition. A ‘chronicle’ can be seen as a HLE — it is expressed in terms of a set of events linked together by time constraints, and, possibly, a set of context constraints. The input language of CRS relies on a reified temporal logic, where propositional terms are related to time-points or other propositional terms. Time is considered as a linearly ordered discrete set of instants. The language includes predicates for persistence, event absence and event repetition. Table 1 presents the CRS predicates. Variables start with an upper case letter while predicates and constants start with a lower-case letter. `?` is the prefix of an atemporal variable. Details about the input language of CRS, and CRS in general, may be found on the web page of the system[2]. The code below, for example, expresses HLE related to vehicle (bus/tram) punctuality in the CRS language:

```
(1)  chronicle punctual[?Id, ?V](T1) {
(2)   event(stop_enter[?Id, ?V, ?S, scheduled], T0)
(3)   event(stop_leave[?Id, ?V, ?S, scheduled], T1)
(4)   T1 > T0
(5)   end - start in [1, 2000]
(6)  }
(7)  chronicle non_punctual[?Id, ?V](T0) {
(8)   event( stop_enter[?Id, ?V, *, late], T0 )
(9)  }
(10) chronicle punctuality_change[?Id, ?V,
      non_punctual](T1) {
(11)  event( punctual[?Id, ?V], T0 )
(12)  event( non_punctual[?Id, ?V], T1 )
(13)  T1 > T0
(14)  noevent( punctual[?Id, ?V], (T0+1, T1) )
(15)  noevent( non_punctual[?Id, ?V], (T0+1, T1) )
(16)  end - start in [1, 20000]
(17) }
```

The atemporal variables of a `chronicle` (HLE) and `event` (LLE or HLE) are displayed in square brackets. `*` denotes that a variable can take any value. Lines (1)–(6) of the above CRS code express a set of conditions in which a vehicle `V` with `Id` is said to be punctual: `V` enters a stop `S` and leaves `S` at the scheduled time. The time-stamp of the ‘punctual’ HLE is the same as that of the ‘stop leave’ subevent (that is, `T1`). The first and the last subevent of the ‘punctual’ HLE, that is, ‘stop enter’ and ‘stop leave’, must take place within 2000 time-points in order to recognise ‘punctual’ (see line (5)). Lines (7)–(9) express one out of several cases in which a vehicle is said to be non-punctual: the vehicle enters a stop after the scheduled time (that is, it is `late`). Lines (10)–(17) express the ‘punctuality change’ HLE: punctuality changes (to non-punctual) when a vehicle that was punctual at an earlier time now is not punctual. Another HLE definition (not shown here to save space) deals with the case in which a vehicle was not punctual earlier and now is punctual.

Note that all events shown in the above CRS code are instantaneous. CRS allows for the representation of durative events. Due to space limitations we do not show here how such events are treated in CRS.

The CRS language does not allow mathematical operators in the constraints of atemporal variables. It is not possible to compute the distance between two entities given

---

[1] `http://www.ict-pronto.org/`

[2] `http://crs.elibel.tm.fr/`

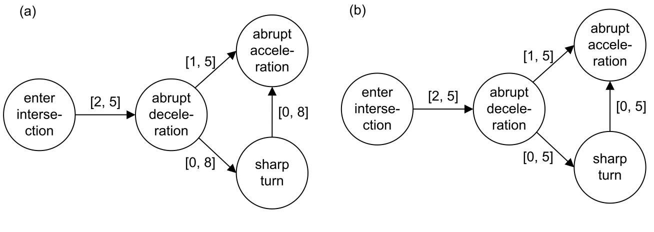

Figure 1: CRS: Constraint Propagation.

their coordinates, for example. In the CTM HLE definition concerning passenger safety, for instance, we cannot express that a vehicle accident or violence within a vehicle is more severe when the vehicle if *far* from a hospital or a police station. Moreover, the CRS language does not allow universal quantification. In CTM, for instance, we cannot define HLE using LLE coming from *all* vehicles (of a particular route).

2.2 Reasoning

Each HLE definition expressed in the CRS language is typically translated to a Temporal Constraint Network (TCN) [16, 11, 12] (see [7], however, for a Petri-Net based semantics of the CRS language). Each subevent of a HLE definition corresponds to a node in the TCN, whereas the temporal constraints between two subevents correspond to the edge linking the nodes expressing the subevents. Figure 1(a), for example, shows a TCN expressing the CTM HLE 'uncomfortable driving'. The subevents of this HLE are 'enter intersection', 'abrupt deceleration', 'sharp turn' and 'abrupt acceleration'. The temporal constraints on these events that, if satisfied, will lead to the recognition of 'uncomfortable driving', are expressed by the edges of the TCN. For example, 'abrupt acceleration' should take place, at the earliest, *1* time-point after the 'abrupt deceleration' LLE and, at the latest, *5* time-points after this LLE. (There are other ways to define 'uncomfortable driving'. This example is presented simply to provide a concrete illustration.)

Before the recognition stage, CRS propagates the constraints of a TCN using an incremental path consistency algorithm in order to produce the least constrained TCN expressing the user constraints. Figure 1(b), for example, shows the TCN for 'uncomfortable driving' after constraint propagation. In this example, the edge between 'abrupt deceleration' and 'sharp turn', and that between 'sharp turn' and 'abrupt acceleration', became [*0*, *5*] due to the temporal constraint [*1*, *5*] between 'abrupt deceleration' and 'abrupt acceleration'. The use of the incremental path consistency algorithm allows for checking the consistency of the temporal constraints of a TCN (details about the use of this algorithm in the context of CRS may be found in [11]). CRS, therefore, detects inconsistent HLE definitions and reports such definitions to the user.

The recognition process of CRS is illustrated in Figure 2 — this figure shows the process of recognising 'uncomfortable driving'. The left part of Figure 2 shows the effects of the arrival of 'enter intersection' at time-point *6*, while the right part of this figure shows the effects of the arrival of 'abrupt deceleration' at time-point *10*. The arrival of 'enter intersection' creates an *instance* of 'uncomfortable driving', that is, a partial instantiation of the definition of this HLE. The horizontal grey lines in Figure 2 show the *temporal windows* of the subevents of 'uncomfortable driving', that is, the possible times in which a subevent may take place without violating the constraints of the 'uncomfortable driving' instance. Upon the arrival of 'enter intersection', the temporal window of 'abrupt deceleration' becomes [*8*, *11*] because, according to the TCN of 'uncomfortable driving' (see Figure 1(b)), 'abrupt deceleration' must take place *2* time-points after the 'enter intersection' LLE at the earliest, and, at the latest, *5* time-points after this LLE. Similarly, the window of 'sharp turn' becomes [*8*, *16*], while that of 'abrupt acceleration' becomes [*9*, *16*]. The occurrence of 'abrupt deceleration' at time-point *10* is integrated in the displayed instance of 'uncomfortable driving' as it complies with the constraints of the instance (that is, 'abrupt deceleration' takes place within its temporal window), and constrains the temporal windows of the (yet) undetected subevents of 'uncomfortable driving' (see the right part of Figure 2).

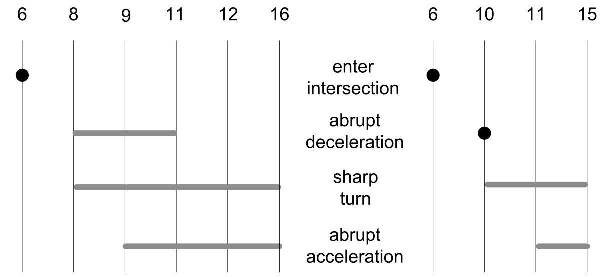

Figure 2: CRS: Recognition Stage.

Using this type of recognition, CRS may report to the user not only a fully recognised HLE, but also a partially recognised HLE, that is, a pending HLE instance. Moreover, CRS may report the events that need to be detected in order to fully recognise a HLE. Such information can be very helpful in various application domains.

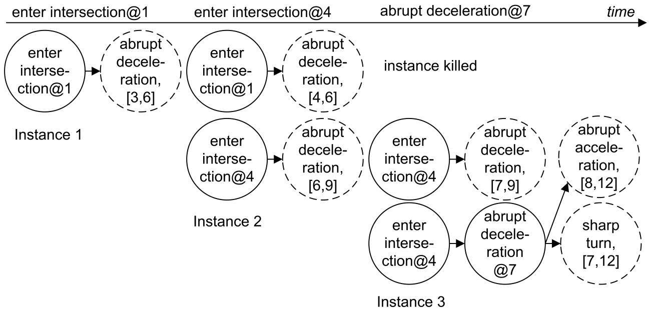

Figure 3: CRS: HLE Instance Management.

Figure 2 shows the evolution of one HLE instance. For each HLE definition more than one instance may be created. Figure 3 illustrates the HLE instance management of CRS — this figure shows instance management concerning 'uncomfortable driving'. The occurrence of 'enter intersection' at time-point *1* creates a new instance of 'uncomfortable driving'. CRS computes the temporal windows of the forthcoming events — for example, 'abrupt deceleration' is expected between [*3*, *6*]. The occurrence of the second 'enter intersection' LLE at time-point *4* creates a new instance of 'uncomfortable driving'. Moreover, the passing of time results in constraining the temporal windows of the forthcoming events of the first instance (for example, the temporal window of 'abrupt deceleration' becomes [*4*, *6*]). Upon the arrival of 'abrupt deceleration' at time-point *7*, CRS makes a copy of the second instance of 'uncomfortable driving', thus creating a third instance of this HLE, and integrates 'abrupt deceleration' in the third HLE instance. CRS keeps the second instance because another 'abrupt deceleration' LLE may take place in the future (more precisely, between 7 and 9), which may lead to another recognition of 'uncomfortable driving'. The first instance of 'uncomfortable driving' is killed at time-point 7 because no 'abrupt deceleration' LLE was detected between $[4, 6]$, and thus it is not possible to satisfy the constraints of this instance any more.

CRS stores all pending HLE instances in trees, one for each HLE definition. Each event occurrence and clock update traverses these trees in order to further develop, or kill, some HLE instances. For K HLE instances, each having n subevents, the complexity of processing each incoming event or a clock update is $\mathcal{O}(Kn^2)$.

Various techniques have been recently developed to reduce the number of created HLE instances and thus improve the efficiency of CRS [12]. One such technique, called *temporal focusing*, can be briefly described as follows. Let's assume that, according to the definition of HLE H, event e_r should take place after event e_f in order to recognise H, e_r is a very rare event and e_f is a frequent event. The frequency of events is determined by an a priori analysis of the application under consideration. In this case CRS stores all incoming e_f events and starts the recognition process, that is, creates a new instance of H, only upon the arrival of an e_r event (the new instance will include e_r and a stored e_f that satisfies the constraints of H). In this way the creation of HLE instances is significantly reduced. (Temporal focusing significantly improves the efficiency of recognising 'uncomfortable driving', for example, as 'enter intersection' is a very frequent LLE, 'abrupt deceleration' is a rare LLE, and 'abrupt deceleration' should take place after 'enter intersection' in order to recognise 'uncomfortable driving'.)

Empirical analysis has shown that CRS can be very efficient and scalable [12]. Recall, however, that CRS is a purely temporal reasoning system.

2.3 Machine Learning

Various approaches have been proposed in the literature for the automated construction of HLE definitions expressed in the CRS language. The proposed approaches include automata-based learning [16], frequency-based analysis of events [14], and inductive logic programming (ILP) [4, 3]. ILP is well-suited to the construction of HLE definitions expressed in the CRS language, as these definitions can be straightforwardly translated into first-order logic descriptions used by ILP systems, and vice-versa. In what follows, therefore, we will present the use of ILP for constructing HLE definitions for CRS.

Note that the approaches mentioned above do not learn the numerical temporal constraints of a HLE definition at the same time as they learn the structure of the HLE, that is, the structure of the TCN expressing the HLE definition. The most common approach is to separate the relational part from the numerical constraints, which are either defined by experts before the relational learning [4], or they are computed from data using simple statistics once the TCN structure is learned [16].

ILP is the combination of inductive machine learning and logic programming. It aims at inducing theories from examples and background knowledge in the form of a first-order logic program. It inherits, from machine learning, the principle of hypothesis induction from data, but its first-order logic representation allows the induction of more expressive theories than classical machine learning approaches, which induce propositional hypotheses. Furthermore, a-priori background knowledge can easily be used to guide learning. The logical elements involved in ILP can be defined as follows (these elements will be shortly explained in terms of HLE definition learning):

- A set of positive examples E^+ and a set of negative examples E^-. These are typically ground facts.
- A set of hypotheses H.
- A background knowledge base B. B and H are sets of clauses of the form $h \leftarrow b_1 \wedge \cdots \wedge b_n$, where the head h and b_i are literals.

ILP searches for hypotheses H such that $B \wedge H \vDash E^+$ (completeness) and $B \wedge H \wedge E^- \nvDash \Box$ (consistency) [26]. The completeness condition guarantees that all positive examples in E^+ can be deduced from H and B. The consistency condition guarantees that no negative examples in E^- can be deduced from H and B. The clauses in H are usually constrained by some form of domain-dependent language bias. A form of language bias that is typically used in ILP is called *mode declarations*. A set M of mode declarations defines a hypothesis space L_M within which $H \subseteq L_M$ must fall. In general, ILP can be seen as a *search problem* over a space of hypotheses, which needs an evaluation function to *estimate* the value of hypotheses under consideration at each step of the search. An exhaustive search is usually impossible, due to the exponential complexity of the space, and thus pruning strategies are employed. The language bias is an effective way of reducing the search space, as it constrains the candidate hypotheses.

Many ILP algorithms have been developed in the literature — such algorithms differ in the way they perform the search, in the way they evaluate the hypotheses, etc. Examples of ILP algorithms include ICL [21], PROGOL[3] and ALEPH[4]. In what follows we illustrate the use of ALEPH for learning HLE definitions expressed in the CRS language (see [3] for a more detailed example).

To learn hypotheses H, expressing HLE definitions concerning punctuality, for example, we use the following mode declarations M and background knowledge B (to save space only a fragment of B is shown):

```
% mode declarations
:- modeh(*, punctual(+id,+vehicle,+float)).
:- modeb(*, event(stop_enter(+id,+vehicle,+stop,
            #respected_time), -float, -evt, -evt)).
:- modeb(*, event(stop_leave(+id,+vehicle,+stop,
            #respected_time), -float, -evt, -evt)).

% background knowledge: LLE for tram tr1
event(stop_enter(tr1,tram,stop1,early),20,init,e1).
event(stop_leave(tr1,tram,stop1,scheduled),20.5,e1,e2).
...
% background knowledge: LLE for bus b2
event(stop_enter(b2,bus,stop4,scheduled),20,init,e1).
event(stop_leave(b2,bus,stop4,scheduled),21,e1,e2).
...
```

A mode declaration is either a head declaration `modeh(r,s)` or a body declaration `modeb(r,s)`, where `s` is a ground literal, the scheme, which serves as a template for literals in the

---

[3] http://www.doc.ic.ac.uk/~shm/progol.html

[4] http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/

head or body of a hypothesis, and `r` is an integer, the recall, which limits how often the scheme is used [33]. In this example the scheme of the head of a hypothesis is `punctual`, while the scheme of the body of a hypothesis is an `event` predicate for `stop_enter` or `stop_leave`. +, -, # express, respectively, input terms, output terms, and ground terms. Note that if we had no prior knowledge about the subevents of `punctual`, we would have written a body declaration for every LLE, stating that any LLE may appear in the body of a hypothesis. The background knowledge base B includes the event narrative used for learning the hypotheses. In this example the narrative consists of a stream of detected LLE — for brevity only `stop_enter` and `stop_leave` are shown above. Notice that a 4-argument `event` predicate is used in the learning procedure. The 4th argument of an `event` predicate represents the `id` of the event that is the 1st argument of the predicate, while the 3rd argument of `event` represents the `id` of the directly temporally preceding event. These two arguments allow for the temporal ordering of events. At the end of the learning procedure the produced hypotheses are translated into the CRS representation shown in Section 2.1

To learn hypotheses H concerning `punctual`, a set of positive examples E^+ and a set of negative examples E^- are given:

```
%E+
punctual(tr1,tram,20.5).    punctual(b2,bus,21).
...
%E-
punctual(tr1,tram,55).      punctual(b2,bus,46).
...
```

Using M, B, E^+ and E^-, ALEPH performs the following operations. First, it selects an example from E^+ to be generalised (for example, `punctual(tr1, tram, 20.5)`). Second, it generates the most specific clause that entails this example with respect to B. Third, it searches for a more general clause than that generated in the previous step, aiming to cover as many positive examples from E^+, without covering any negative examples from E^-. Fourth, it adds the clause to H, removing redundant clauses and restarting with a new example from E^+ until E^+ is empty.

In practice, the examples used to induce a hypothesis in H, as well as the event narrative that is part of B, may be noisy. In order to facilitate learning under such conditions, ILP systems relax the consistency and completeness requirements, allowing some negative examples to be deduced from H and B and some positive ones to not be covered. (An approach that has been specifically developed for learning hypotheses in noisy environments is presented in Section 4).

The result of ILP in this example comprises the following:

```
[Rule 1]
punctual(Id,V,T2) :-
   event(stop_enter(Id,V,S,early),T1,E0,E1),
   event(stop_leave(Id,V,S,scheduled),T2,E1,E2).
[Rule 2]
punctual(Id,V,T2) :-
   event(stop_enter(Id,V,S,scheduled),T1,E0,E1),
   event(stop_leave(Id,V,S,scheduled),T2,E1,E2).
```

These rules express the HLE definition concerning a punctual vehicle. After the end of the ILP procedure, the numerical temporal constraints are added using simple statistics as mentioned earlier. Moreover, the generated hypotheses are translated into the CRS language. For instance, the second rule shown above is translated into the first chronicle presented in Section 2.1.

Learning HLE definitions that have other HLE as subevents is performed in a similar manner. In this case, however, one would have to add to the background knowledge base B a HLE narrative, as opposed to a LLE narrative. To learn the definition of the 'punctuality change' HLE, for example, B would have to include a narrative of `punctual` and `non_punctual` HLE.

3. THE EVENT CALCULUS

The Event Calculus (EC), introduced by Kowalski and Sergot [20], is a many-sorted, first-order predicate calculus for representing and reasoning about events and their effects. EC was not originally developed for event recognition, but has recently been used for this task. EC is typically expressed as a logic (Prolog) program (see [13], however, for a Java implementation of EC, and [24] for an implementation using satisfiability solvers). Various dialects of EC have been proposed in the literature (for event recognition). In Section 3.1 we present a high-level review of the expressiveness of EC as a logic programming language, in Section 3.2 we present a concrete implementation of this formalism, while in Section 3.3 we present techniques for automatically constructing an EC logic program.

Table 2: Predicates of the Event Calculus.

Predicate	Meaning
happensAt$(E,\ T)$	Event E is occurring at time T
happensFor$(E,\ I)$	I is the list of the maximal intervals during which event E takes place
initially$(F = V)$	The value of fluent F is V at time 0
holdsAt$(F = V,\ T)$	The value of fluent F is V at time T
holdsFor$(F = V,\ I)$	I is the list of the maximal intervals for which $F = V$ holds continuously
initiatedAt$(F = V,\ T)$	At time T a period of time for which $F = V$ is initiated
terminatedAt$(F = V, T)$	At time T a period of time for which $F = V$ is terminated

3.1 Representation

The time model of EC is often linear and it may include real numbers or integers. Where F is a *fluent* — a property that is allowed to have different values at different points in time — the term $F = V$ denotes that fluent F has value V. Boolean fluents are a special case in which the possible values are true and false. Informally, $F = V$ holds at a particular time-point if $F = V$ has been *initiated* by an event at some earlier time-point, and not *terminated* by another event in the meantime.

An *event description* in EC includes rules that define the event occurrences (with the use of the happensAt and happensFor predicates), the effects of events (with the use of the initiatedAt and terminatedAt predicates), and the values of the fluents (with the use of the initially, holdsAt and holdsFor predicates). Table 2 summarises the main EC predicates. Variables start with an upper-case letter while predicates and constants start with a lower-case letter.

EC has built-in rules for holdsAt and holdsFor, that is, for computing the value of a fluent at a particular time and for computing the maximal intervals in which a fluent has a particular value (there are EC dialects with additional built-in rules for more expressive temporal representation [23]). A partial specification of holdsAt, for example, is given below:

$$\begin{aligned}&\mathsf{holdsAt}(F\,{=}\,V,\ T) \leftarrow\\ &\quad \mathsf{initiatedAt}(F\,{=}\,V,\ T_s),\ T_s \leq T,\\ &\quad \mathsf{not\ broken}(F\,{=}\,V,\ T_s,\ T)\end{aligned} \tag{1}$$

$$\begin{aligned}&\mathsf{broken}(F\,{=}\,V,\ T_s,\ T) \leftarrow\\ &\quad \mathsf{terminatedAt}(F\,{=}\,V,\ T_e),\ T_s \leq T_e \leq T\end{aligned} \tag{2}$$

not represents 'negation by failure'. The above rules state that $F = V$ holds at T if $F = V$ has been initiated at T_s, where $T_s \leq T$, and not 'broken', that is, terminated, in the meantime. The events that initiate/terminate a fluent are represented in the body of initiatedAt and terminatedAt. The interested reader is referred to the cited papers for alternative implementations of holdsAt, and implementations of holdsFor (in the following section we sketch one implementation of holdsFor).

The EC rules below express a set of conditions in which a vehicle is said to be punctual/non-punctual:

$$\begin{aligned}&\mathsf{happensAt}(punctual(Id,\ V),\ DT) \leftarrow\\ &\quad \mathsf{happensAt}(stop\_enter(Id,\ V, S, scheduled),\ AT),\\ &\quad \mathsf{happensAt}(stop\_leave(Id,\ V, S, scheduled),\ DT),\\ &\quad 1 \leq DT{-}AT \leq 2000\end{aligned} \tag{3}$$

$$\begin{aligned}&\mathsf{happensAt}(non\_punctual(Id,\ V),\ AT) \leftarrow\\ &\quad \mathsf{happensAt}(stop\_enter(Id,\ V, \_, late),\ AT)\end{aligned} \tag{4}$$

All events in the above rules are instantaneous and thus they are represented by means of happensAt. Punctuality *change* may be expressed in EC as follows:

$$\mathsf{initially}(punctuality(\_, \_) = punctual) \tag{5}$$

$$\begin{aligned}&\mathsf{initiatedAt}(punctuality(Id,\ V) = punctual,\ \ T) \leftarrow\\ &\quad \mathsf{happensAt}(punctual(Id,\ V),\ \ T)\end{aligned} \tag{6}$$

$$\begin{aligned}&\mathsf{initiatedAt}(punctuality(Id,\ V) = non\_punctual,\ \ T) \leftarrow\\ &\quad \mathsf{happensAt}(non\_punctual(Id,\ V),\ \ T)\end{aligned} \tag{7}$$

$$\begin{aligned}&\mathsf{happensAt}(punctuality\_change(Id,\ V,\ Value),\ \ T) \leftarrow\\ &\quad \mathsf{holdsFor}(punctuality(Id,\ V) = \ Value,\ \ I),\\ &\quad (T, \_) \in I,\ \ T \neq 0\end{aligned} \tag{8}$$

We have defined an auxiliary fluent, *punctuality*, that records the time-points in which a vehicle is (non-)punctual. The fluent *punctuality* is defined by rules (5)–(7). Rule (8) expresses the definition of the HLE 'punctuality change'. This rule uses the EC built-in implementation of holdsFor to compute the maximal intervals for which a vehicle is continuously (non-)punctual. Punctuality changes at the first time-point of each of these intervals (see the last line of rule (8)).

Note that, depending on the requirements of the user (city transport officials, in the CTM example), *punctuality* may itself be a HLE, as opposed to an auxiliary construct. In general, a HLE may not necessarily be treated as an EC event. In some cases it is more convenient to treat a HLE as an EC fluent. In the case of a durative HLE H, for example, treating H as a fluent and using the built-in holdsFor to compute the intervals of H, may result in a more succinct representation than treating H as an EC event and developing domain-dependent rules for happensFor to compute the intervals of H.

The availability of the full power of logic programming is one of the main attractions of employing EC as the temporal formalism. Paschke et al. [30, 31, 29], for instance, have used the power of logic programming to develop predicates for event recognition that are used in the context of EC. These predicates include implementations of the following event operators: sequence, mutual exclusivity, concurrency, aperiodicity, and so on. In general, the availability of logic programming allows EC HLE definitions to include not only complex temporal constraints (EC is at least as expressive as the CRS language with respect to temporal representation), but also complex atemporal constraints. For example, it is straightforward to develop in Prolog a predicate computing the distance between two entities.

Logic programming, not including an EC implementation, has been used frequently for event recognition. A notable example can be found in [35]. A benefit of EC, in comparison to pure Prolog, is that EC has built-in rules for complex temporal representation, including the formalisation of inertia, events with delayed effects, continuous change [23], etc, which help considerably the development of HLE definitions.

3.2 Reasoning

Several implementations of the EC built-in rules have been proposed in the literature. Reasoning in EC is often performed at *query-time*, that is, the incoming LLE are logged without processing them, and reasoning about the LLE log is performed when a query, concerning the recognition of HLE, is submitted. In most cases query-time reasoning is not accompanied by caching techniques, that is, the outcome of query computation is not stored. Although this type of reasoning may be acceptable for retrospective event recognition, that is, recognition performed after the operation of the application under consideration (for example, recognition performed at the end of the day in order to evaluate the performance of public transportation), the absence of caching techniques does not allow for run-time event recognition, that is, recognition performed during the operation of the monitored application (for instance, recognition performed during the day in order to detect, at real-time, incidents affecting the smooth operation of public transportation). To perform run-time event recognition using query-time reasoning, one would have to repeatedly query EC (say every 5 seconds). If the outcome of query computation (the intervals of the recognised HLE) is not stored, reasoning would be performed on *all* detected LLE, as opposed to the LLE detected between two consecutive query times. Consequently, recognition time would substantially increase over time. (In retrospective recognition, querying about the intervals of a HLE is performed once, so there is considerably less need to store the outcome of query computation.)

To overcome the above limitation a cached version of the Event Calculus has been proposed: the so-called *Cached Event Calculus (CEC)* [6]. Reasoning in CEC is not performed at query-time, but at *update-time*: CEC infers and *stores* all consequences of LLE as soon as they arrive. Query processing, therefore, amounts to retrieving the appropriate HLE intervals from the memory.

Note that caching does not necessarily imply update-time reasoning. Caching techniques may be implemented for

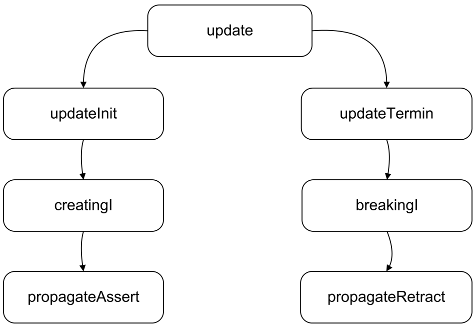

Figure 4: The Cached Event Calculus (from [6]).

query-time reasoning.

Figure 4 shows the main modules of CEC. Each new LLE is entered into the database using *update*. *updateInit* and *updateTermin* are then called to manage fluents that are initiated and, respectively, terminated by the LLE. Recall that a fluent may represent a HLE or it may represent a context variable used in the definition of a HLE. *updateInit* may call *creatingI* to create a new maximal interval for a fluent. *updateTermin* may call *breakingI* to clip a maximal interval of a fluent. The modules *propagateAssert* and *propagateRetract* deal with the non-chronological arrival of LLE, that is, the arrival of a LLE that happened (was detected) before some of the already acquired LLE. When a maximal interval (or part of it) of a fluent is retracted, or asserted, as a result of the occurrence of a LLE that arrived in a non-chronological manner, the update has to be propagated to the fluents whose validity may rely on such an interval. The retraction or assertion of an interval $[T_1, T_2]$ in which a fluent has a particular value modifies the context of events occurring at time-points belonging to this interval, and possibly invalidates (or activates) the effects of these events. *propagateAssert* and *propagateRetract* may recursively activate the process of creating or breaking maximal intervals, by means of calling *creatingI* and *breakingI*. To avoid clutter in Figure 4, however, we do not show the information flow between *propagateAssert*, *propagateRetract* and the remaining CEC modules.

The complexity of update processing (inferring the consequences of events) in CEC, measured in terms of accesses to happensAt and holdsFor Prolog facts (happensAt facts represent the incoming LLE while holdsFor facts represent cached fluent intervals, including HLE intervals), is $\mathcal{O}(n^{L_{fw}+3})$, where n is the number of initiating and terminating events for any fluent, and L_{fw} is the maximum number of propagations of fluent interval assertions and retractions — see discussion above on *propagateAssert* and *propagateRetract*. The complexity of query processing (retrieving cached fluent intervals) in CEC is $\mathcal{O}(n)$. Details about the complexity analysis of CEC may be found in [6].

The efficiency of CEC has been reported to be adequate for certain application domains [5]. In practice, where delayed LLE are considered only if the delay does not exceed a certain threshold, the complexity of update processing is considerably less than the worst-case complexity presented above. Moreover, ways to improve the efficiency of CEC have been identified [1]. Note, however, that caching in CEC concerns only HLE represented as fluents, and thus needs to be extended to cater for HLE represented as EC events (such as, for example, *punctuality_change* — see Section 3.1).

3.3 Machine Learning

Since EC event descriptions are typically expressed as logic programs, Inductive Logic Programming (ILP) methods are an obvious candidate for constructing domain-dependent rules representing HLE definitions. As discussed in Section 2.3, ILP can be used to induce hypotheses from examples. For instance, to learn the definition of the HLE *punctual*, one has to provide positive examples E^+ and negative examples E^- for *punctual* using the happensAt predicate, and a background knowledge base B including a LLE narrative. The learnt hypotheses will be of the form of rules (3) and (4). In general, learning hypotheses for predicates for which examples are available (such as $\mathsf{happensAt}(punctual(Id, V), T)$), that is, 'observation predicate learning' (OPL) [25], may be achieved using ILP techniques as shown in Section 2.3.

Automatically constructing an EC logic program often includes learning hypotheses for predicates for which examples are *not* available, which implies that induction cannot be directly applied to produce the required hypotheses. Consider, for instance, the case in which we need to learn the definition of the CTM HLE 'reducing passenger satisfaction', we require to represent this HLE as a fluent in terms of initiatedAt (because, say, we expect that such a representation would be succinct), and the available examples for learning this HLE are given only in terms of holdsAt. In such a case, abduction may be combined with induction in order to produce the required hypotheses. Abduction may produce ground initiatedAt rules, using the examples expressed by means of holdsAt and the EC built-in rules, such as (1) and (2), relating initiatedAt and holdsAt. Then, induction may generalise the outcome of abduction.

Various approaches have been proposed in the literature for combining abduction with induction in order to learn a logic program. In what follows we will briefly describe the XHAIL system [33] that has been recently developed for this task, and has been used for learning EC programs. The learning technique of XHAIL is based on the construction and generalisation of a preliminary ground hypothesis, called a Kernel Set, that bounds the search space in accordance to user specified language and search bias. XHAIL follows a three-stage process. First, abduction is used to compute the head literals of a Kernel Set. Second, deduction is used to compute the body literals of the Kernel Set. Third, induction is used to generalise the clauses of the Kernel Set. Each stage is specified as an executable abductive logic programming (ALP) task. Consequently, all three stages may be implemented using any ALP reasoner or answer set solver.

We will illustrate the use of XHAIL by showing how it may be used to learn the definition of the 'reducing passenger satisfaction' HLE. As mentioned above, we require to represent this HLE as a fluent in terms of initiatedAt, while the available examples are given in terms of holdsAt. The input to XHAIL for learning this HLE is a background knowledge base B including the built-in EC rules and a LLE narrative, and a set of positive and negative examples E such as:

$$\mathsf{holdsAt}(reducing\_passenger\_satisfaction(b_1, bus) = \mathsf{true}, 8)$$
$$\mathsf{not\ holdsAt}(reducing\_passenger\_satisfaction(b_1, bus) = \mathsf{true}, 6)$$

The first phase of XHAIL, that is, the abductive phase, computes ground initiatedAt atoms. The computed atoms $\Delta = \bigcup_{i=1}^{n} \alpha_i$ are such that E is entailed by B and Δ. Below is an atom produced by the abductive phase of XHAIL:

$$\mathsf{initiatedAt}(reducing\_passenger\_satisfaction(b_1, bus) = \mathsf{true}, 8)$$

Recall that $\mathsf{initiatedAt}$ and $\mathsf{holdsAt}$ are related by the EC built-in rules (see rule (1), for instance). Each abduced $\mathsf{initiatedAt}$ atom will go in the head of a Kernel Set clause.

The second phase of XHAIL, that is, the deductive phase, computes a ground program $K = \bigcup_{i=1}^{n} \alpha_i \leftarrow \delta_i^1, \dots, \delta_i^{m_i}$ such that every δ_i^j, where $1 \leq i \leq n$ and $1 \leq j \leq m_i$, is entailed by B and Δ. In other words, the second phase adds body literals in the clauses of the Kernel Set K. In this example, a body literal may be *any* LLE, that is, we have no prior knowledge concerning what affects passenger satisfaction, and any fluent expressing in-vehicle conditions such as temperature, noise level and passenger density. The user restricts the range of possible body literals of the Kernel Set by means of mode declarations. Below is a clause of the produced Kernel Set K:

$$
\begin{array}{l}
\mathsf{initiatedAt}(\mathit{reducing\_passenger\_satisfaction}(b_1, \mathit{bus}) = \mathsf{true},\\
\quad 8) \leftarrow\\
\qquad \mathsf{happensAt}(\mathit{passenger\_density\_change}(b_1, \mathit{bus}, \mathit{high}),\ 8),\\
\qquad \mathsf{holdsAt}(\mathit{temperature}(b_1, \mathit{bus}) = \mathit{very\_warm}),\ 8),\\
\qquad \mathsf{holdsAt}(\mathit{noise\_level}(b_1, \mathit{bus}) = \mathit{high}),\ 8)
\end{array}
$$

A passenger density increase initiates a period of time for which passenger satisfaction is reducing, provided that in-vehicle temperature is very warm and noise level is high. Note that this clause concerns a particular time-point (8).

The third phase of XHAIL, that is, the inductive phase, computes a theory H that subsumes K and entails E with respect to B. Below is a clause of the computed theory H:

$$
\begin{array}{l}
\mathsf{initiatedAt}(\mathit{reducing\_passenger\_satisfaction}(\mathit{Id},\ V) = \mathsf{true},\\
\quad T) \leftarrow\\
\qquad \mathsf{happensAt}(\mathit{passenger\_density\_change}(\mathit{Id},\ V, \mathit{high}),\ T),\\
\qquad \mathsf{holdsAt}(\mathit{temperature}(\mathit{Id},\ V) = \mathit{very\_warm}),\ T)
\end{array}
$$

Noise level is not included in the above clause because it did not prove to be a determining factor of the reduction of passenger satisfaction.

The proposed combination of abduction and induction has been applied to relatively small and noise-free applications [33]. As mentioned in Section 2.3, the examples (annotated HLE) used to induce a hypothesis, as well as the event narrative (annotated or detected LLE or HLE) that is part of the background knowledge base, may be noisy. Next we present an approach that has been specifically developed for learning and reasoning about hypotheses in noisy environments.

4. MARKOV LOGIC

Event recognition systems often have to deal with the following issues [1, 35]: incomplete LLE streams, erroneous LLE detection, inconsistent LLE and HLE annotation, and a limited dictionary of LLE and context variables. These issues may compromise the quality of the (automatically or manually) constructed HLE definitions, as well as HLE recognition accuracy. In this section we review Markov Logic Networks that consider uncertainty in representation, reasoning and machine learning, and, consequently, address, to a certain extent, the aforementioned issues.

4.1 Representation

In order to deal with uncertainty, probabilistic graphical models can be used in event recognition. Sequential graphical models such as Dynamic Bayesian Networks [27] and Hidden Markov Models [32] are useful for modelling HLE definitions representing event sequences. Event recognition with such models is usually performed through maximum likelihood estimation on the LLE sequences. For large-scale applications with complex events that involve long-term dependencies and hierarchical structure, sequential models have been extended into more complex variants. For instance, the model in [18] captures long-term dependencies, in [28] hierarchical events are represented, and the model in [19] can be applied to structured data. However, these models have restricted temporal representation and most of them allow only for sequential relations between events. Moreover, they do not naturally incorporate domain-specific knowledge.

On the other hand, logic-based formalisms, such as first-order logic, can compactly represent complex event relations, but do not naturally handle uncertainty. Assume, for example, a first-order logic knowledge base expressing HLE definitions. A *possible world* assigns a truth value to each possible ground atom. A missed LLE or an erroneous LLE detection, violating even a single formula of the knowledge base, may result in a zero-probability world.

The research communities of Statistical Relational Learning and Probabilistic Inductive Logic Programming have proposed a variety of methods [8, 9, 15] that combine concepts from first-order logic and probabilistic models. This approach is adopted by Knowledge-Based Model Construction (KBMC) methods, where a logic-based language is used to generate a propositional graphical model on which probabilistic inference is applied [9]. Markov Logic Networks (MLN) [10, 34] is a recent and rapidly evolving KBMC framework, which provides a variety of reasoning and learning algorithms[5], and has recently been used for event recognition [2, 38]. The main concept behind MLN is that the probability of a world increases as the number of formulas it violates decreases. Therefore, a world violating formulas becomes less probable, but not impossible as in first-order logic. Syntactically, each formula F_i in Markov logic is represented in first-order logic and it is associated with a weight w_i. The higher the value of the weight, the stronger the constraint represented by F_i. Semantically, a set of Markov logic formulas (F_i, w_i) represents a probability distribution over possible worlds.

Consider, for example, the formulas below expressing a simplified version of the definition of the 'uncomfortable driving' CTM HLE:

$$
\begin{array}{l}
\mathit{abrupt\_movement}(Id, V, T) \leftarrow\\
\qquad \mathit{abrupt\_acceleration}(Id, V, T) \ \vee\\
\qquad \mathit{abrupt\_deceleration}(Id, V, T) \ \vee\\
\qquad \mathit{sharp\_turn}(Id, V, T)
\end{array}
\tag{9}
$$

$$
\begin{array}{l}
\mathit{uncomfortable\_driving}(Id, V, T_2) \leftarrow\\
\qquad \mathit{enter\_intersection}(Id, V, T_1) \ \wedge\\
\qquad \mathit{abrupt\_movement}(Id, V, T_2) \ \wedge\\
\qquad \mathit{before}(T_1, T_2)
\end{array}
\tag{10}
$$

Variables, starting with upper-case letters, are universally quantified unless otherwise indicated. Predicates and constants start with a lower-case letter. The definition of *uncomfortable_driving* is simplified here in order to facilitate the presentation of MLN reasoning techniques that will be given in the following section. According to the above formulas, *uncomfortable_driving* is defined in terms another HLE, *abrupt_movement*, which is in turn defined

---

[5] A system implementing MLN reasoning and learning algorithms may be found at `http://alchemy.cs.washington.edu/`

in terms of the *abrupt_acceleration*, *abrupt_deceleration* and *sharp_turn* LLE. *before* is a simple predicate comparing two time-points. Formulas (9) and (10), expressing the definitions of the aforementioned HLE, have real-valued positive weights.

MLN facilitate a mixture of *soft constraints* and *hard constraints* in a HLE knowledge base, where hard constraints correspond to formulas with infinite weight values. Hard constraints can be used to capture domain-specific knowledge or facts. For example, a bus is driven only by one driver at a time. Soft constraints, on the other hand, can be used to capture imperfect logical statements and their weights provide their confidence value. Strong weights are given to formulas that are almost always true. For instance, we may assign a strong weight to formula (10), as it is true most of the time. Respectively, weak weights may be assigned to formulas that describe exceptions. For example, we may assign a weak weight to the formula stating that 'unsafe driving' is recognised when we have 'abrupt movement' — normally a '*very* abrupt movement' leads to the recognition of 'unsafe driving'.

4.2 Reasoning

A MLN L is a template that produces a ground Markov network $M_{L,C}$ by grounding all its formulas F, using a finite set of constants $C = c_1, ...c_{|C|}$. All formulas are translated into *clausal form* and the weight of each formula is equally divided among its clauses. For different sets of constants, the same MLN L will produce different ground Markov networks, but all will have certain regularities in structure and parameters — for example, all groundings of a clause will have the same weight. Each node in a $M_{L,C}$ is represented by a boolean variable and corresponds to a possible grounding of a predicate that appears in L. Each subset of ground predicates, appearing in the same ground clause, are connected to each other and form a clique in $M_{L,C}$. Each clique is associated with the corresponding weight w_i of a clause and a feature. The value of the feature is 1 when the ground clause is true, otherwise it is 0.

A ground Markov network $M_{L,C}$, therefore, is composed of a set X of random variables (ground predicates). A *state* $x \in \mathcal{X}$ of $M_{L,C}$ represents a possible world, as it assigns truth values to all random variables X. A probability distribution over states is specified by the ground Markov network $M_{L,C}$, and represented as follows:

$$P(X = x) = \frac{1}{Z} exp\left(\sum_i^{|F_y|} w_i n_i(x)\right) \quad (11)$$

$F_y \subseteq F$ is the set of clauses, w_i is the weight of the i-th clause, $n_i(x)$ is the number of true groundings of the i-th clause in x, and Z is the partition function used for normalisation, that is, $Z = \sum_{x \in \mathcal{X}} exp(P(X = x))$, where $\mathcal{X}$ is the set of all possible states.

In event recognition, the detected LLE provide the constants C that are necessary for producing ground Markov networks expressing a knowledge base of HLE definitions. Consider, for example, the HLE definitions expressed by formulas (9) and (10), and a LLE stream including *enter_intersection*(*tr$_0$*, *tram*, *100*), *sharp_turn*(*tr$_0$*, *tram*, *101*), etc. Figure 5 shows a fragment of the ground Markov network. Recall that formulas are translated into clausal form — formula (9), for example, is translated into three clauses, while formula (10) is translated into a single clause. Predicates, appearing in the same ground clause, are connected to each other in the network and form a clique.

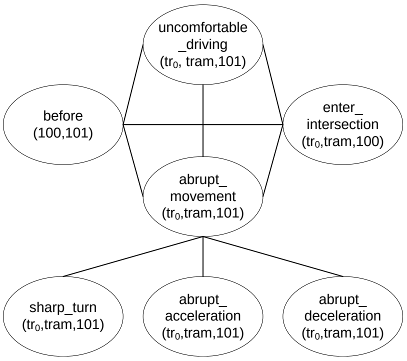

Figure 5: Ground Markov Network.

In order to demonstrate how the probability of each state of a ground Markov network is computed, we will assume that the partial network of Figure 5 is the complete Markov network. We will calculate the probability of two states, x_1 and x_2, where both states assign the same truth values to all predicates except *uncomfortable_driving*(*tr$_0$*, *tram*, *101*). More precisely, in both states *enter_intersection*(*tr$_0$*, *tram*, *100*), *abrupt_acceleration*(*tr$_0$*, *tram*, *101*), *abrupt_movement*(*tr$_0$*, *tram*, *101*), *before*(*100*, *101*) are true, and *sharp_turn*(*tr$_0$*, *tram*, *101*) and *abrupt_deceleration*(*tr$_0$*, *tram*, *101*) are false. *uncomfortable_driving*(*tr$_0$*, *tram*, *101*) is true in x_1 and false in x_2. In x_1 every clause has one true grounding. In state x_2 only the three clauses of formula (9) have true groundings — each one has one true grounding. Using eq. (11) we compute the following:

$$P(X = x_1) = \frac{1}{Z} exp(\tfrac{1}{3} w_1 \cdot 1 + \tfrac{1}{3} w_1 \cdot 1 + \tfrac{1}{3} w_1 \cdot 1 + w_2 \cdot 1) = \frac{1}{Z} e^{w_1 + w_2}$$

$$P(X = x_2) = \frac{1}{Z} exp(\tfrac{1}{3} w_1 \cdot 1 + \tfrac{1}{3} w_1 \cdot 1 + \tfrac{1}{3} w_1 \cdot 1 + w_2 \cdot 0) = \frac{1}{Z} e^{w_1}$$

w_1 is the weight of formula (9) — w_1 is equally divided to each of the three clauses of this formula — while w_2 is the weight of formula (10). According to the above results, a state in which the 'uncomfortable driving' HLE and its subevents have all been recognised is more probable than a state in which the subevents of 'uncomfortable driving' have been recognised while this HLE does not hold.

Sensors may detect LLE with certainty or with a degree of confidence — usually probability. In the former case, the LLE are simply boolean variables that are given directly to the MLN as evidence. In the latter case, the detected LLE are added to the MLN knowledge base as clauses having weights proportional to their detection probability. The negation of these clauses is also provided with a weight proportional to the complement of the LLE detection probability [38]. For example, if the LLE *sharp_turn*(*tr$_0$*, *tram*, *20*) is detected with probability 0.7, *sharp_turn*(*tr$_0$*, *tram*, *20*) and $\neg$*sharp_turn*(*tr$_0$*, *tram*, *20*) will be added to the MLN, having weights $w_1 \propto 0.7$ and $w_2 \propto 0.3$ respectively. In this manner, the detected LLE propagate their detection probability to the MLN.

Complete grounding of MLN, even for simple knowledge bases, results in complex and large networks. For this reason, only the minimal required network is constructed. In particular, evidence variables are used to separate the network into regions, allowing some variables to be removed from the network, as they cannot influence reasoning. For example, if *abrupt_movement*(tr_0, *tram*, *101*) is known to be true (from previous inference), then the nodes corresponding to *abrupt_acceleration*(tr_0, *tram*, *101*), *sharp_turn*(tr_0, *tram*, *101*) and *abrupt_deceleration*(tr_0, *tram*, *101*) can safely be removed from the network, as they cannot influence the reasoning process. In addition, further efficiency can be gained by employing (a) *lazy inference* methods that ground predicates as and when needed [10, Section 3.3] or (b) *lifted inference* [10, Section 3.4].

Event recognition in MLN involves querying a ground network about HLE. In particular, the set X of random variables of a ground network can be partitioned as $X = Q \cup E \cup H$, where Q is the set of query variables, that is, the HLE of interest, E is the set of evidence variables, that is, the detected LLE, and H is the set of the remaining variables with unknown value — also known as hidden variables — which are auxiliary constructs of a HLE definition. There are two basic types of inference in graphical models, *maximum a posteriori* (MAP) inference and the computation of *conditional probabilities.* For both types, a variety of exact and approximate probabilistic inference methods exist in the literature. However, probabilistic inference in MLN may become infeasible due to the size and the complexity of the ground network. For this reason, several methods combining ideas from logical and probabilistic inference have been proposed [10, Chapter 3].

Event recognition queries require conditional inference, that is, computing the probability that a predicate holds given some evidence. Given a MLN and some evidence $E = e$, a conditional query is specified as follows:

$$P(Q \mid E = e, H) = \frac{P(Q, E = e, H)}{P(E = e, H)} \qquad (12)$$

Q are the query predicates, H are the hidden variables, and the numerator and denominator may be computed using eq. (11). We may be interested in finding out, for instance, the trams that are driven in an uncomfortable manner given a LLE stream. In this case, the set of query variables Q includes only *uncomfortable_driving*(Id, V, T), the set of observed LLE that forms E includes, among others:

enter_intersection(tr_0, *tram*, *100*)
abrupt_acceleration(tr_0, *tram*, *101*)
sharp_turn(tr_{24}, *tram*, *100*)

and the set of hidden variables H includes, among others:

abrupt_movement(tr_0, *tram*, *101*)
abrupt_movement(tr_{24}, *tram*, *101*)

Given eq. (12) we may compute the probability of each grounding of *uncomfortable_driving*.

Eq. (12) can be efficiently approximated by sampling methods, such as Markov Chain Monte Carlo (MCMC) algorithms — for example, Gibbs sampling. The conditional probability in eq. (12), can be computed by a MCMC that rejects all moves to states where $E = e$ does not hold. In MCMC, the successive sample depends only on the current sample and not on its predecessors. In fact, each sample differs only marginally from its predecessor — for example, by the truth value of a predicate. Following this idea, MCMC performs a random walk in the state space. Recall that each ground predicate is connected with other predicates in the network only when they appear together in some grounding of a clause. Consequently, the probability of a ground predicate depends only on the probabilities of its neighbour predicates in the network, and is independent from the rest of the network. The set of neighbouring predicates, called *Markov blanket*, is exploited by the Gibbs sampling algorithm for probabilistic inference. For example, the Markov blanket of *abrupt_movement* in the network shown in Figure 5, is composed of *uncomfortable_driving*, *before*, *enter_intersection*, *sharp_turn*, *abrupt_acceleration* and *abrupt_deceleration*. The algorithm has the tendency to get stuck in local maxima and, therefore, the more samples are generated, the more accurate the estimation becomes.

Due to the combination of logic with probabilistic models, inference must handle both deterministic and probabilistic dependencies. Deterministic or near-deterministic dependencies are formed from formulas with infinite and strong weights respectively. MCMC, being a pure statistical method, can only handle probabilistic dependencies. Deterministic dependencies create isolated regions in the state space by introducing zero-probability (impossible) states. Similarly, near-deterministic dependencies create regions that are difficult to cross, that is, contain states with near zero-probability. To overcome this problem and deal with both types of dependency, satisfiability testing is combined with MCMC [10, Section 3.2]. In particular, satisfiability testing helps MCMC move between isolated and difficult-to-cross regions.

4.3 Learning

Learning a MLN involves estimating the weights of the network and/or the first-order formulas forming the network structure, given a set of training data, that is, LLE annotated with HLE. Weight learning in MLN is performed by optimising a likelihood function, which is a statistical measure of how well the probabilistic model (MLN) fits the training data. In particular, weights can be learned by either generative or discriminative estimation [10, Section 4.1]. Generative learning attempts to optimise the joint distribution of all variables in the model. In contrast, discriminative learning attempts to optimise the conditional distribution of a set of outputs, given a set of inputs.

Generative estimation methods search for weights that optimise the likelihood function, given a HLE knowledge base and training data. Learning can be performed by a standard gradient-descent optimisation algorithm. However, it has been shown that computing the likelihood and its gradient is intractable [34]. For this reason, the optimisation of the pseudo-likelihood function is used instead, which is the product of the probabilities of the ground predicates, conditioned on their neighbours in the network (Markov blanket) [34]. In particular, if x is a state of a ground network and x_l is the truth value of the l-th ground predicate X_l, the pseudo-likelihood of x, given weights w, is:

$$log\ P_w^*(X = x) = \textstyle\sum_{l=1}^{n} log\ P_w(X_l = x_l \mid MB_x(X_l)) \qquad (13)$$

$MB_x(X_l)$ represents the truth values of the ground predicates in the Markov blanket of X_l. Thus, computation can be performed very efficiently, even in domains with millions of ground predicates, as it does not require inference over the complete network.

The pseudo-likelihood function assumes that each ground predicate's Markov blanket is fully observed, and does not exploit information obtained from longer-range dependencies in the network. In some cases such an implementation may lead to poor results. Consider, for example, the HLE definitions of *abrupt_movement* and *uncomfortable_driving* (see formulas (9) and (10)). As mentioned earlier, the Markov blanket of *abrupt_movement* includes the *sharp_turn*, *abrupt_acceleration* and *abrupt_deceleration* LLE, while the Markov blanket of *uncomfortable_driving* includes *abrupt_movement*. Assume that the training dataset includes a stream of the aforementioned LLE, and has annotations only for *uncomfortable_driving*, that is, there is no annotation for *abrupt_movement*. In this case pseudo-likelihood will give poor results with respect to the *uncomfortable_driving* HLE. This is due to the fact that pseudo-likelihood will only use information from the Markov blanket of this HLE, making an assumption about the absent annotation of *abrupt_movement* (usually the closed-world assumption), and will not exploit the information provided by the *abrupt_acceleration*, *abrupt_deceleration* and *sharp_turn* LLE of the training dataset.

In event recognition we know *a priori* which set of variables will form the evidence and which ones will concern queries — evidence variables represent LLE while query variables represent HLE. In the usual case, where we aim to recognise the latter given the former, it is preferable, as will be explained below, to learn the weights discriminatively by maximising the conditional likelihood function. In particular, if we partition the variables of the domain into a set of evidence variables E and a set of query variables Q, then the conditional likelihood function is defined as follows:

$$P(Q=q \mid E=e) = \frac{1}{Z_e} exp\left(\sum_i w_i n_i(e,q)\right) \qquad (14)$$

Z_e normalises over all states consistent with the evidence e, and $n_i(e,q)$ is the number of true groundings of the i-th clause in the training dataset. It has been shown that learning weights discriminatively can lead to higher predictive accuracy than generative learning [36]. This is partly due to the fact that, in contrast to the pseudo-likelihood function, conditional likelihood can exploit information from longer-range dependencies. Similar to the likelihood function, conditional likelihood requires inference. However, there is one key difference: conditioning on the evidence in a Markov network reduces significantly the number of likely states. Therefore, inference takes place on a simpler model and the computational requirements are reduced. Optimisation techniques for computing conditional likelihood in the context of MLN may be found in [10, Section 4.1].

In addition to weight learning, the structure of a MLN can be learned from training data. In principle, the structure of a MLN can be learned in two stages, using any ILP method, as presented in Section 2.3, and then performing weight learning. However, separating the two learning tasks in this way may lead to suboptimal results, as the first optimisation step (ILP) needs to make assumptions about the weight values, which have not been optimised yet. Better results can be obtained by combining structure learning with weight learning in a single stage.

A variety of structure learning methods have been proposed for MLN, which optimise a likelihood function and create the structure by employing ILP techniques. In brief, these methods can be classified into top-down and bottom-up methods. Top-down structure learning [10, Section 4.2] starts from an empty or existing MLN and iteratively constructs clauses by adding or revising a single predicate at a time, using typical ILP operations and a search procedure (for example, beam search). However, as the structure of a MLN may involve complex HLE definitions, the space of potential top-down refinements may become intractable. For this reason, bottom-up structure learning can be used instead, starting from training data and searching for more general hypotheses [10, Section 4.2]. This approach usually leads to a more specialised model, following a search through a manageable set of generalisations.

5. SUMMARY AND OPEN ISSUES

We presented three representative logic-based approaches to event recognition. All approaches assume as input a stream of time-stamped low-level events (LLE), which are being processed to detect high-level events (HLE) of interest. We illustrated the use of the three approaches, drawing examples from the domain of city transport management.

Being based on logic, all three approaches benefit from a formal semantics, a variety of inference mechanisms, and methods for learning a knowledge base from data. As a result, compared to commonly used procedural methods, logic-based ones facilitate efficient development and management of event definitions, which are clearly separated from the generic inference mechanism and their consistency is verifiable. Furthermore, recent logic-based methods appear to be sufficiently mature and scalable to be used in industrial applications.

The presented Chronicle Recognition System (CRS) has been specially developed for event recognition and is the choice of preference for efficient, purely temporal recognition. CRS was developed with the aim to support only temporal reasoning and thus a line of future work concerns its extension with atemporal reasoning.

The Event Calculus (EC) provides a more generic and expressive representation of events, taking advantage of the full power of logic programming on which it is based. Thus, EC supports complex temporal as well as atemporal representation and reasoning. A line of further work concerns the optimisation of the reasoning of EC for run-time event recognition. Caching techniques, in particular, should be investigated, supporting all types of HLE representation.

Markov Logic Networks (MLN) are useful for handling noisy event streams, as they combine the strengths of logical and probabilistic inference. Their use for event recognition has been very limited so far and there are many issues that need to be resolved still, such as the incorporation and use of numerical temporal constraints in MLN inference.

Finally, a number of challenging issues remain open in learning event definitions. Examples of such issues are the use of abduction to handle partial supervision that is commonly available for event recognition and the simultaneous optimisation of numerical parameters (for example, weights and temporal constraints) and the logical structure of the knowledge base expressing HLE definitions.

Acknowledgments

This work has been partially funded by EU, in the context of the PRONTO project (FP7-ICT 231738).

6. REFERENCES

[1] A. Artikis, A. Skarlatidis, G. Paliouras, P. Karampiperis, and C. Spyropoulos. First knowledge of event definitions and reasoning algorithms for event recognition. In *Deliverable 4.1.1 of the EU-funded FP7 PRONTO project (FP7-ICT 231738)*, 2010.

[2] R. Biswas, S. Thrun, and K. Fujimura. Recognizing activities with multiple cues. In *Workshop on Human Motion*, LNCS 4814, pages 255–270. Springer, 2007.

[3] L. Callens, G. Carrault, M.-O. Cordier, É. Fromont, F. Portet, and R. Quiniou. Intelligent adaptive monitoring for cardiac surveillance. In *ECAI*, pages 653–657, 2008.

[4] G. Carrault, M. Cordier, R. Quiniou, and F. Wang. Temporal abstraction and inductive logic programming for arrhyhtmia recognition from electrocardiograms. *Artificial Intelligence in Medicine*, 28:231–263, 2003.

[5] L. Chittaro and M. Dojat. Using a general theory of time and change in patient monitoring: Experiment and evaluation. *Computers in Biology and Medicine*, 27(5):435–452, 1997.

[6] L. Chittaro and A. Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 12(3):359–382, 1996.

[7] C. Choppy, O. Bertrand, and P. Carle. Coloured petri nets for chronicle recognition. In *Proceedings of Conference on Reliable Software Technologies*, volume LNCS 5570, pages 266–281. Springer, 2009.

[8] L. De Raedt and K. Kersting. Probabilistic inductive logic programming. *Probabilistic Inductive Logic Programming*, pages 1–27, 2008.

[9] R. de Salvo Braz, E. Amir, and D. Roth. A survey of first-order probabilistic models. In D. E. Holmes and L. C. Jain, editors, *Innovations in Bayesian Networks*, volume 156 of *Studies in Computational Intelligence*, pages 289–317. Springer, 2008.

[10] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence.* Morgan & Claypool Publishers, 2009.

[11] C. Dousson. Alarm driven supervision for télécommunication network II — on-line chronicle recognition. *Annales des Telecommunication*, 51(9–10):501–508, 1996.

[12] C. Dousson and P. L. Maigat. Chronicle recognition improvement using temporal focusing and hierarchisation. In *IJCAI*, pages 324–329, 2007.

[13] A. Farrell, M. Sergot, M. Sallé, and C. Bartolini. Using the event calculus for tracking the normative state of contracts. *International Journal of Cooperative Information Systems*, 4(2–3):99–129, 2005.

[14] F. Fessant, F. Clérot, and C. Dousson. Mining of an alarm log to improve the discovery of frequent patterns. In *Industrial Conference on Data Mining*, pages 144–152, 2004.

[15] L. Getoor and B. Taskar. *Introduction to statistical relational learning.* The MIT Press, 2007.

[16] M. Ghallab. On chronicles: Representation, on-line recognition and learning. In *Principles of Knowledge Representation and Reasoning*, pages 597–606, 1996.

[17] A. Hakeem and M. Shah. Learning, detection and representation of multi-agent events in videos. *Artificial Intelligence*, 171(8–9):586–605, 2007.

[18] S. Hongeng and R. Nevatia. Large-scale event detection using semi-hidden markov models. In *Proceedings of Conference on Computer Vision*, pages 1455–1462. IEEE, 2003.

[19] K. Kersting, L. De Raedt, and T. Raiko. Logical hidden markov models. *Journal of Artificial Intelligence Research*, 25(1):425–456, 2006.

[20] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–96, 1986.

[21] W. V. Laer. *From Propositional to First Order Logic in Machine Learning and Data Mining.* PhD thesis, K. U. Leuven, 2002.

[22] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley, 2002.

[23] R. Miller and M. Shanahan. The event calculus in a classical logic — alternative axiomatizations. *JETAI*, 4(16), 2000.

[24] E. Mueller. *Commonsense Reasoning.* Morgan Kaufmann, 2006.

[25] S. Muggleton and C. Bryant. Theory completion using inverse entailment. In *ILP*, volume LNCS 1866, pages 130–146. Springer, 2000.

[26] S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[27] K. Murphy. *Dynamic bayesian networks: representation, inference and learning.* PhD thesis, University of California, Berkeley, 2002.

[28] N. Nguyen, D. Phung, S. Venkatesh, and H. Bui. Learning and detecting activities from movement trajectories using the hierarchical hidden Markov model. In *Proceedings of Conference on Computer Vision and Pattern Recognition*, 2005.

[29] A. Paschke. ECA-RuleML: An approach combining ECA rules with temporal interval-based KR event logics and transactional update logics. Technical Report 11, Technische Universität München, 2005.

[30] A. Paschke and M. Bichler. Knowledge representation concepts for automated SLA management. *Decision Support Systems*, 46(1):187–205, 2008.

[31] A. Paschke, A. Kozlenkov, and H. Boley. A homogeneous reaction rule language for complex event processing. In *Proceedings of Workshop on Event driven Architecture, Processing and Systems*, 2007.

[32] L. Rabiner and B. Juang. A tutorial on hidden Markov models. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[33] O. Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3), 2009.

[34] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[35] V. Shet, J. Neumann, V. Ramesh, and L. Davis. Bilattice-based logical reasoning for human detection. In *Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

[36] P. Singla and P. Domingos. Discriminative training of markov logic networks. In *Proceedings of AAAI*, pages 868–873. AAAI Press, 2005.

[37] M. Thonnat. Semantic activity recognition. In *Proceedings of ECAI*, pages 3–7, 2008.

[38] S. D. Tran and L. S. Davis. Event modeling and recognition using markov logic networks. In *Proceedings of Computer Vision Conference*, pages 610–623, 2008.