Logic-Based Event Recognition

Alexander Artikis[1], Anastasios Skarlatidis[1 3], François Portet[2] and Georgios Paliouras[1]
[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 their large data volumes into operational knowledge. This requirement may be addressed by employing event recognition systems that detect events/activities of special significance within an organisation, given streams of 'low-level' information that is very difficult to be utilised by humans. Consider, for example, the recognition of attacks on nodes of a computer network given the TCP/IP messages, the recognition of suspicious trader behaviour given the transactions in a financial market, and the recognition of whale songs given a symbolic representation of whale sounds. Various 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 have 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. We illustrate the reviewed approaches with the use of a real-world case study: event recognition for city transport management.

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 need 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 (Luckham 2002) — accept as input a stream of time-stamped low-level events (LLE). A LLE is the result of applying a computational derivation process to some other event, such as an event coming from a sensor. Using LLE as input, event recognition systems identify high-level events (HLE) of interest — collections of events that satisfy some pattern[1]. Consider, for example, the recognition of attacks on nodes of a computer network given the TCP/IP messages, the recognition of suspicious trader behaviour given the transactions in a financial market, and the recognition of whale songs given a symbolic representation of whale sounds.

Numerous event recognition systems have been proposed in the literature — see (Luckham 2002; Vu et al. 2003; Lv et al. 2005; Arasu et al. 2006; Hakeem and Shah 2007; Thonnat 2008; Etzion and Niblett 2010) for a few examples and (Cugola and Margara 2011; Paschke and Kozlenkov 2009) for two recent surveys. 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.

[1] Pottebaum and Marterer (2010) discuss the relationship between the terms 'low-level event' and 'high-level event' and other terms proposed in the literature, including the glossary of the Event Processing Technical Society (Luckham and Schulte 2008).

Logic-based event recognition systems exhibit a formal, declarative semantics, in contrast to other types of recognition system that often exhibit an informal and/or procedural semantics. As pointed out in (Paschke 2005), 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 recognition systems may be found in (Paschke 2005).

Non-logic-based event recognition systems have proven to be, overall, more efficient than logic-based ones 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 systems 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[2] system, for example, has been used in enterprise event processing networks.

The 'definition' of a HLE imposes temporal and, possibly, atemporal constraints on its 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), the Event Calculus (EC), and Markov Logic Networks (MLN). CRS is a purely temporal reasoning 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. EC, which has also been used for event recognition, 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 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.

Running Example: City Transport Management

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[3], an event recognition system is being developed with the aim to support the management of public transport — see Figure 1. 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 and 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 (Artikis, Kukurikos et al. 2011).

[2] `http://www.prova.ws`
[3] `http://www.ict-pronto.org/`

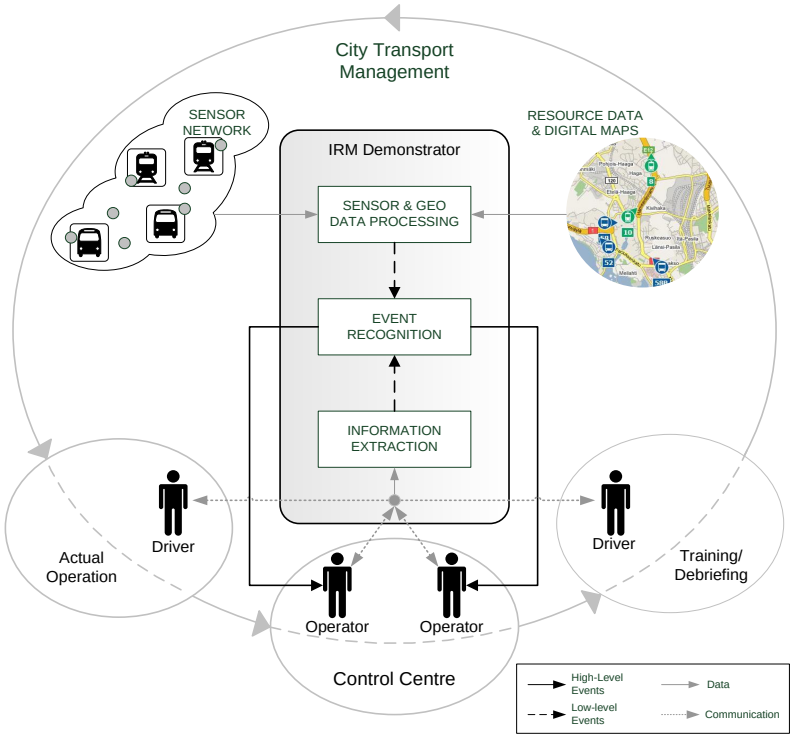

Fig. 1: Event Recognition for City Transport Management.

2 Chronicle Recognition

Chronicle recognition systems are temporal reasoning systems developed for efficient, run-time HLE — chronicle — recognition. A chronicle is expressed in terms of a set of events linked together by time constraints and, possibly, a set of context constraints. A number of implementations have been developed for chronicle recognition. In this section we will present the chronicle recognition system (hereafter CRS) of Dousson and colleagues[4] (Dousson 2002; Dousson and Maigat 2006; Dousson and Maigat 2007). CRS is an extension of IxTeT (Ghallab and Alaoui 1989; Dousson et al. 1993; Ghallab 1996; Dousson 1996), a version of which was marketed and tested for the supervision of gas turbines, aircraft turbines, and electrical power networks. CRS has been applied to numerous application domains such as cardiac monitoring (Callens et al. 2008), intrusion detection (Morin and Debar 2003) and mobility management (Dousson et al. 2007) in computer networks, and distributed diagnosis of web services (Le Guillou et al. 2008).

In the following section we present the input language of CRS, in Section 2.2 we present various reasoning algorithms of this system, and in Section 2.3 we present techniques for automatically constructing HLE definitions in the CRS language.

2.1 Representation

A chronicle can be seen as a HLE — as mentioned above, 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

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

Table 1: Predicates of CRS.

Predicate	Meaning
`event(E, T)`	Event `E` takes place at time-point `T`
`event(F:(?V1,?V2), T)`	An event takes place at time-point `T` changing the value of attribute `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 attribute `F` from `V1` to `V2`
`hold(F:?V, (T1,T2))`	The value of attribute `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`)

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. 'Attributes' represent context information. Attributes and events (also called 'messages' in the CRS language) may have any number of parameters. Details about the input language of CRS, and CRS in general, can be found on the web page of the system[4].

The code below, for example, expresses HLE related to vehicle (bus/tram) punctuality in the CRS language:

```
(1)  chronicle punctual[?Id, ?VehicleType](T2) {
(2)    event(stop_enter[?Id, ?VehicleType, ?StopId, scheduled], T1)
(3)    event(stop_leave[?Id, ?VehicleType, ?StopId, scheduled], T2)
(4)    T2 - T1 in [1, 2000]
(5)  }
(6)  chronicle non_punctual[?Id, ?VehicleType](T1) {
(7)    event(stop_enter[?Id, ?VehicleType, *, late], T1)
(8)  }
(9)  chronicle punctuality_change[?Id, ?VehicleType, non_punctual](T2) {
(10)   event(punctual[?Id, ?VehicleType], T1)
(11)   event(non_punctual[?Id, ?VehicleType], T2)
(12)   T2 - T1 in [1, 20000]
(13)   noevent(punctual[?Id, ?VehicleType], (T1+1, T2))
(14)   noevent(non_punctual[?Id, ?VehicleType], (T1+1, T2))
(15) }
```

The atemporal variables of a `chronicle` (HLE) and an `event` (LLE or HLE) are displayed in square brackets. `*` denotes that a variable can take any value. Lines (1)–(5) of the above CRS code express a set of conditions in which a vehicle of a particular type (represented by `VehicleType`) and `Id` is said to be punctual: the vehicle enters a stop and leaves the same stop (represented by `StopId`) at the scheduled time. The time-stamp of the 'punctual' HLE is the same as that of the 'stop leave' subevent (that is, `T2`). 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 (4)). Lines (6)–(8) 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`). 'non-punctual' (respectively 'punctual') is defined as a disjunction of 'stop enter' and 'stop leave' LLE satisfying certain conditions. Disjunction is expressed in the CRS language with the use of multiple `chronicle`s (explicit representation of disjunction is not allowed because

it makes reasoning in CRS NP-complete — a presentation of the reasoning techniques of CRS is given in the following section). For simplicity we do not show here the `chronicle`s expressing the other cases in which a vehicle is said to be non-punctual (respectively punctual).

Lines (9)–(15) of the above code fragment 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 (similar to the one shown above) deals with the case in which a vehicle was not punctual earlier and now is punctual.

Both quantitative and qualitative temporal constraints can be represented in the CRS language, the latter being replaced by numerical constraints during compilation — for instance, a constraint of the form `T1 > T0` is translated to `T1 - T0 in` $[1,\infty)$. More details about the compilation stage of CRS are given in the following section. Note that the CRS language allows events not to be completely ordered. Consider the following code fragment:

```
event(abrupt_acceleration[?Id, ?VehicleType], T1)
event(sharp_turn[?Id, ?VehicleType], T2)
T2-T1 in [-3, 8]
```

According to the above constraints, 'sharp turn' may take place before, at the same time, or after 'abrupt acceleration'.

All events shown in the above code fragments are instantaneous. CRS does not allow for the explicit representation of durative events. One may implicitly represent the interval/duration of such an event in the CRS language by representing two instantaneous events, one indicating the time-point in which the durative event starts taking place, and one indicating the time-point in which it stops occurring. The CRS treatment of durative events allows the representation of the interval relations of a restricted interval algebra (Vilain and Kautz 1986; Nebel and Bürckert 1995), but does not support all of Allen's (1983) interval relations.

Often in the literature durative events are not even implicitly represented — they are treated as if they occur at an atomic instant. It has been pointed out (Paschke 2005) that such a treatment leads to logical problems and unintended semantics for several event algebra operators — such operators facilitate the development of HLE definitions.

CRS is a temporal reasoner and, as such, it does not support mathematical operators on the constraints of atemporal variables. It is not possible to compute the distance between two entities given 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 support universally quantified conditions. In CTM, for instance, we cannot define HLE using LLE coming from *all* vehicles (of a particular route).

Although the CRS language is limited in the aforementioned ways, it has proven to be expressive enough for numerous application domains, some of which were mentioned in the beginning of Section 2.

2.2 Reasoning

Each HLE definition expressed in the CRS language is typically translated to a Temporal Constraint Network (TCN) (Dechter et al. 1991; Ghallab 1996; Dousson 1996; Dousson and Maigat 2007) (see (Choppy et al. 2009), 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 determine the edge between the nodes expressing the subevents. Figure 2(a), for example, shows the TCN expressing the CTM HLE 'uncomfortable driving'. The subevents of this HLE are 'approach

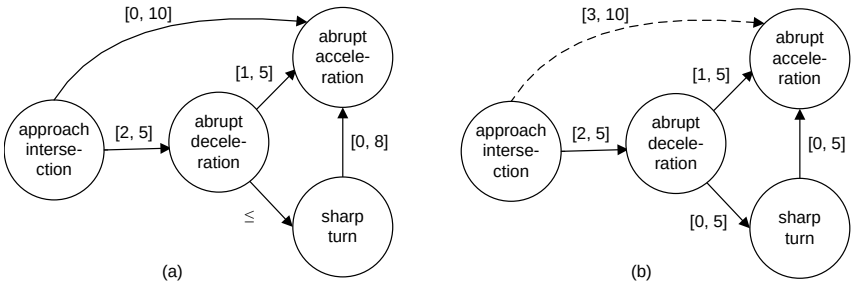

Fig. 2: Temporal Constraint Network.

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 deceleration' should take place, at the earliest, *2* time-points after the 'approach intersection' LLE and, at the latest, *5* time-points after this LLE. Briefly, a vehicle is said to be driven in an uncomfortable manner if, within a specified time period, it approaches an intersection, and then decelerates abruptly, turns sharply and accelerates abruptly. (There are other ways of defining 'uncomfortable driving'. This example is presented simply to provide a concrete illustration.) The CRS code of this simplified definition of 'uncomfortable driving' may be found below:

```
(1)  chronicle uncomfortable_driving[?Id, ?VehicleType](T4) {
(2)    event(approach_intersection[?Id, ?VehicleType], T1)
(3)    event(abrupt_deceleration[?Id, ?VehicleType], T2)
(4)    event(sharp_turn[?Id, ?VehicleType], T3)
(5)    event(abrupt_acceleration[?Id, ?VehicleType], T4)
(6)    T2 - T1 in [2, 5]
(7)    T4 - T2 in [1, 5]
(8)    T4 - T3 in [0, 8]
(9)    T4 - T1 in [0, 10]
(10)   T2 <= T3
(11) }
```

During the off-line compilation stage, CRS propagates the constraints of a TCN using an incremental path consistency algorithm (Mackworth and Freuder 1985), in order to produce the least constrained TCN expressing the user constraints. Figure 2(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', becomes $[0, 5]$ due to the temporal constraint $[1, 5]$ between 'abrupt deceleration' and 'abrupt acceleration'. The constraint $[3, 10]$ (dashed in Figure 2(b)) is removed because it is redundant with respect to the two other constraints $[2, 5]$ and $[1, 5]$ from which it can be completely deduced.

The use of the incremental path consistency algorithm allows for checking the consistency of the temporal constraints of a TCN — see (Dousson 1996) for details. CRS, therefore, detects inconsistent HLE definitions at compile-time and reports the inconsistencies to the user.

Once the least constrained TCN expressing the user constraints have been compiled, HLE recognition may commence. The recognition process of CRS is illustrated in Figure 3 — this figure shows the process of recognising 'uncomfortable driving'. The left part of Figure 3 shows the effects of the arrival of 'approach 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 'approach intersection' creates an *instance* of 'uncomfortable driving', that is, a partial instantiation of the definition of this HLE. The horizontal grey lines in Figure

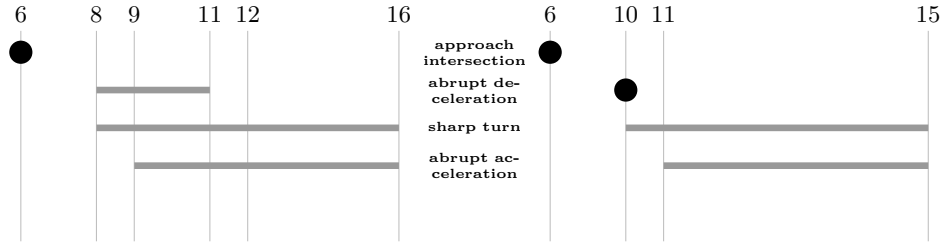

Fig. 3: HLE Instance Recognition.

3 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 'approach intersection', the temporal window of 'abrupt deceleration' becomes $[8, 11]$ because, according to the TCN of 'uncomfortable driving' (see Figure 2(b)), 'abrupt deceleration' must take place *2* time-points after the 'approach intersection' LLE at the earliest, and, at the latest, *5* time-points after this LLE. Similarly, the temporal 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 into 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 further the temporal windows of the (yet) undetected subevents of 'uncomfortable driving' (see the right part of Figure 3).

Using this type of recognition, CRS may report to the user not only a fully recognised HLE, but also a partially recognised one, that is, a pending HLE instance. Moreover, CRS may report the events that need to be detected in order to fully recognise a HLE. These types of information have proven to be very helpful in various application domains (see, for example, (Gao et al. 2009)).

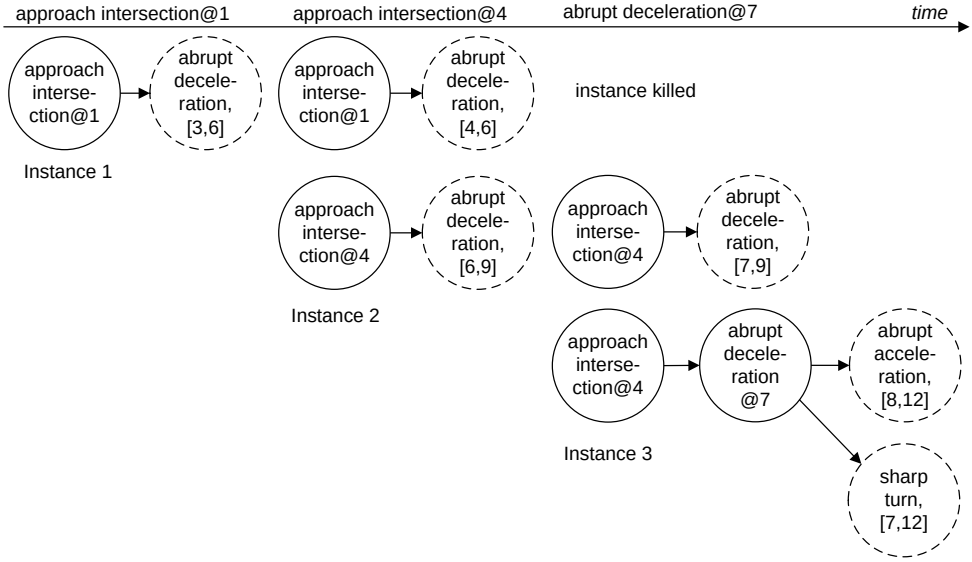

Fig. 4: HLE Instance Management.

Figure 3 shows the evolution of one HLE instance. For each HLE definition more than one instance may be created. Figure 4 illustrates the instance management of CRS using the example of the 'uncomfortable driving' HLE. The occurrence of 'approach intersection' at time-point *1* creates a new instance of

Table 2: Event Recognition in CRS.

Input LLE	**Output HLE**
`event(approach_intersection[b5, bus], 5)`	
`event(abrupt_deceleration[b5, bus], 7)`	
`event(sharp_turn[b5, bus], 8)`	
`event(stop_enter[b5, bus, s8, scheduled], 15)`	
`event(stop_leave[b5, bus, s8, late], 17)`	
`event(abrupt_acceleration[b5, bus], 19)`	
`event(approach_intersection[b5, bus], 36)`	
`event(abrupt_deceleration[b5, bus], 38)`	
`event(sharp_turn[b5, bus], 38)`	
`event(abrupt_acceleration[b5, bus], 39)`	`event(uncomfortable_driving[b5, bus], 39)`
`event(abrupt_acceleration[b5, bus], 42)`	`event(uncomfortable_driving[b5, bus], 42)`
`event(abrupt_acceleration[b5, bus], 55)`	
...	

'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 'approach 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' into 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.

In the examples presented in this section we assume that events arrive in a timely manner. Such an assumption is made to simplify the presentation. We will discuss shortly the consequences of the delayed arrival of events.

Table 2 illustrates event recognition in CRS using a larger LLE narrative. More precisely, this table shows an example LLE narrative concerning a particular vehicle, and the 'uncomfortable driving' HLE that are recognised given this narrative. By time-point *19* all sub-events of the 'uncomfortable driving' HLE are detected. However, this HLE is not recognised because the detected LLE do not satisfy the temporal constraints of the HLE definition. The occurrence of 'abrupt acceleration' at time-point *19* is too late; at that time there are no pending instances of 'uncomfortable driving' (the last instance was killed at time-point *14*).

The first recognition of 'uncomfortable driving' takes place at time-point *39*. The occurrence of 'abrupt acceleration' at that time is integrated into the pending instance of 'uncomfortable driving' that consists of the occurrence of 'approach intersection' at time-point *36*, 'abrupt deceleration' and 'sharp turn' at time-point *38*. This instance is duplicated before integrating the occurrence of 'abrupt acceleration' at *39* because it may lead to another recognition of 'uncomfortable driving' in the future. Indeed, at time-point *42* 'uncomfortable driving' is recognised again: the occurrence of 'abrupt acceleration' at time-point *42* is integrated into the aforementioned pending instance.

Unlike the occurrences of 'abrupt acceleration' at time-points *39* and *42*, the occurrence of this LLE

at time-point *55* does not lead to the recognition of 'uncomfortable driving'. At this time-point there is no pending instance of 'uncomfortable driving', as the last pending instance was killed at time-point *44*.

CRS stores all pending HLE instances in trees, one for each HLE definition. At the arrival of a new event, and at a clock update, CRS 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 $\mathcal{O}(Kn^2)$.

To allow for the delay in the information exchange between distributed components, such as the LLE detection component and the HLE recognition component, CRS processes LLE arriving in a non-timely manner — the user specifies the maximum allowed delay for LLE. In this way, CRS may process LLE arriving in a non-chronological order, that is, process LLE that happened (were detected) before some of the already acquired LLE. This feature, however, affects the efficiency of CRS: CRS delays the time at which a pending HLE instance must be killed due to the possibility of a late arrival of a subevent of the HLE. In other words, the number K of HLE instances reduces at a slower rate.

Various techniques have been proposed for reducing the number K of HLE instances that are generated, and thus improving the efficiency of CRS. Bounding the temporal distance between the first subevent and the last subevent of a HLE (see, for example, the definition of `punctual` in Section 2.1) is one way to reduce HLE instance duplication. Moreover, there may be HLE that cannot have two completed instances overlapping in time or share the occurrence of an event. In this case, when a HLE instance is completed (the HLE is recognised) all its pending instances must be removed.

A recently developed technique for reducing the number of generated HLE instances, and, therefore, improving the efficiency of CRS, is called *temporal focusing* (Dousson and Maigat 2007). This technique can be briefly described as follows. Let's assume that, according to the definition of HLE Γ, event e_r should take place after event e_f in order to recognise Γ, 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 Γ, 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 Γ. In this way the number of generated HLE instances is significantly reduced.

Temporal focusing significantly improves the efficiency of recognising 'uncomfortable driving', for example, as 'approach intersection' is a very frequent LLE, 'abrupt deceleration' is a rare LLE, and 'abrupt deceleration' should take place after 'approach intersection' in order to recognise 'uncomfortable driving'.

Empirical analysis has shown that CRS can be very efficient and scalable (Dousson and Maigat 2007).

Although the CRS language is fairly comprehensible, synthesising and generalising expert temporal knowledge, which can be highly application-dependent, is not a trivial task. Therefore, methods for automatically generating and refining HLE definitions from data are highly desirable. In the following section we review machine learning techniques that have been used for constructing HLE definitions in the CRS language.

2.3 Machine Learning

Various approaches have been proposed in the literature for the automated construction of HLE definitions expressed in the CRS language. One of the earliest approaches is the automata-based learning method of Ghallab (1996). Briefly, this method, which is inspired from learning techniques used in syntactical pattern recognition, learns automata from positive and negative HLE examples. Based on the learnt automata, discriminative 'skeleton models' are generated, containing only events and constraints, without context information. These skeleton models have to be completed by the human user with context information as well as quantitative temporal constraints.

Another line of research for the automated construction of HLE definitions concerns the use of unsu-

pervised learning techniques. One such technique is the frequency-based analysis of sequences of events — see, for example, (Dousson and Duong 1999; Yoshida et al. 2000; Fessant et al. 2004; Hirate and Yamana 2006; Vautier et al. 2007; Álvarez et al. 2010). Most of the approaches adopting this technique rely on an extended version of the *Apriori* algorithm (Mannila et al. 1997) to discover frequent sequences of events — the temporal distance between the events of a frequent sequence is computed using lower and higher frequency thresholds. A well-known approach for HLE definition discovery is the FACE system of Dousson and Duong (1999). FACE uses an algorithm for incremental generation of definitions of HLE that are frequent in event narratives/logs. Fessant et al. (2004) pointed out that FACE is very memory consuming in the presence of large event narratives. To address this issue, Fessant et al. proposed a pre-processing phase based on a Kohonen's self-organising map (Kohonen 2001) to extract the most 'interesting' sub-narratives of events before the data mining phase of FACE. Frequency-based analysis is a promising approach for discovering unknown event patterns in databases or logs. However, these approaches are limited to propositional learning. Moreover, they may not be adapted to learning definitions of HLE that are not frequent in data — in some applications the recognition of such HLE is of utmost importance.

A common technique for learning HLE definitions in a supervised manner concerns the use of inductive logic programming (ILP) (Muggleton 1991) — see, for example, (Carrault et al. 2003; Callens et al. 2008). ILP is well-suited to the construction of HLE definitions expressed in the CRS language because, among others, the CRS HLE definitions can be straightforwardly translated into the logic programming representation used by ILP systems, and vice-versa — this translation is illustrated below. In what follows, therefore, we will present the use of ILP for constructing HLE definitions for CRS.

To illustrate the translation of CRS HLE definitions into logic programming consider, for example, the definition of 'uncomfortable driving' in logic programming (this definition was presented in the CRS language in the previous section):

```
uncomfortable_driving(T4, Id, VehicleType) ←
    approach_intersection(T1, Id, VehicleType),
    abrupt_deceleration(T2, Id, VehicleType),
    sharp_turn(T3, Id, VehicleType),
    abrupt_acceleration(T4, Id, VehicleType),
    T2-T1 >= 2, T2-T1 =< 5,
    T4-T2 >= 1, T4-T2 =< 5,
    T4 >= T3, T4-T3 =< 8,
    T4 >= T1, T4-T1 =< 10,
    T2 =< T3
```

A `chronicle` ('uncomfortable driving', in this example), as well as every `event` in the definition of a `chronicle` (here 'approach intersection', 'abrupt deceleration', 'sharp turn' and 'abrupt acceleration'), can be translated into a predicate whose arguments are the `chronicle`/`event` occurrence time (`T1`, for example) and the `chronicle`/`event` parameters (for example, `Id` and `VehicleType`). Temporal constraints in the CRS language can be translated into arithmetic expressions — for instance, `T2-T1 in [2, 5]` can be translated into `T2-T1 >= 2, T2-T1 =< 5`. In general, any `chronicle` definition can be translated into a Horn clause whose head is the predicate representing the `chronicle`, and whose body is the conjunction of predicates representing the `event`s of the `chronicle` and arithmetic expressions constraining the occurrence of these `event`s.

ILP is the combination of inductive machine learning and logic programming. It aims at inducing theories from examples 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

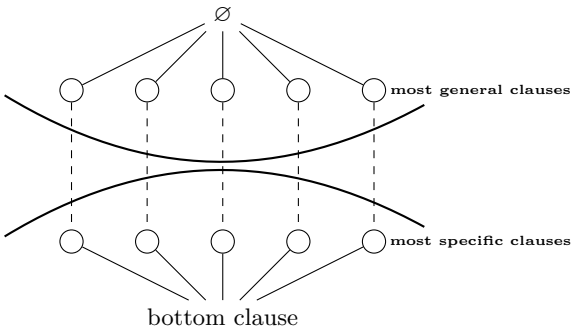

Fig. 5: Version Space.

hypotheses. Furthermore, a-priori background knowledge, such as human expertise, can easily be used to guide learning. ILP has proven adequate in learning from very small datasets, especially in the presence of strong prior knowledge, while making good use of large volumes of data, where available. This also includes constructing theories that capture exceptional cases in datasets, where exceptions are a small minority of a dataset. A detailed account of ILP may be found in (Dzeroski and Lavrac 2001; Konstantopoulos et al. 2008), for example.

The logical elements involved in ILP can be defined as follows:

- A set of positive examples E^+ and a set of negative examples E^-. These are typically ground facts.
- A hypothesis language $\mathcal{L}_H$, from which hypotheses H are constructed.
- A background knowledge base B. B and H are sets of clauses/rules of the form $\alpha \leftarrow l_1, \ldots, l_n$ where α is a 'head' literal and each l_i is a 'body' literal.

ILP searches for hypotheses $H \subseteq \mathcal{L}_H$, such that $B \wedge H \models E^+$ (completeness) and $B \wedge H \wedge E^- \not\models \square$ (consistency) (Muggleton and Raedt 1994). The completeness condition guarantees that all positive examples in E^+ can be deduced from H and B. The consistency condition guarantees that no negative example in E^- can be deduced from H and B.

Different ILP methods adopt different strategies to induce hypotheses H. A simple and common approach starts by selecting a positive example e^+ from E^+, constructing a conjunctive first-order clause $h \in \mathcal{L}_H$ that entails e^+, with respect to the background knowledge B, but does not entail any example from E^-. All positive examples covered by h are then removed and the same procedure is iterated over the remaining positive examples to produce a new clause. Learning a clause h can be seen as walking through the space of clauses of $\mathcal{L}_H$ that entail e^+, with respect to B. Such a space, also known as *version space*, is shown in Figure 5. The empty clause, that is, the most general clause, is at the top of the space, and the most specific clause from $\mathcal{L}_H$ that entails e^+ with respect to B — the 'bottom clause' — is at the bottom of the space. According to the strategy that is adopted, the search may be general-to-specific or specific-to-general. In a general-to-specific strategy, the search will start with the empty clause and will try to specialise it, for instance, by adding conjuncts that cover the positive example and help exclude negative ones. The opposite happens in a specific-to-general strategy that starts with the bottom clause. An exhaustive search is usually impossible, due to the exponential complexity of the version space, and thus pruning and heuristics must be employed to guide the search. For instance, during the search, if a clause does not cover the positive example under consideration (respectively covers a negative example) then it is meaningless to specialise (respectively generalise) this clause further, because it would not improve its coverage. When using a search method that allows backtracking, such as best-first search, a small list of the 'best' clauses generated so far is maintained, sorted according to their coverage. In this manner, search can backtrack directly to the next best option and continue from that point in the version space.

Reducing the version space and guiding more efficiently the search are two of the main challenges of ILP. One way to address these challenges is by acting on *inductive bias*. In ILP three types of bias are used (Nédellec et al. 1996): the *search bias* which specifies how the space is walked; the *validation bias* which determines the criteria according to which a candidate hypothesis (clause) is evaluated; and the *language bias* which restricts $\mathcal{L}_H$. A form of language bias that is typically used in ILP is called *mode declarations*. This particular form of bias will be illustrated presently.

Many ILP algorithms have been developed in the literature. These algorithms differ in, among others, the way they perform the search, and the way they bias learning. Examples of ILP algorithms include FOIL (Quinlan and Cameron-Jones 1995), PROGOL[5](Muggleton and Bryant 2000), ICL[6] (Laer 2002) and ALEPH[7]. In what follows we illustrate the use of ALEPH — a frequently used ILP algorithm — for learning HLE definitions expressed in the CRS language. A detailed example of learning CRS HLE definitions using ILP may be found in (Carrault et al. 2003).

To learn hypotheses H expressing HLE definitions of 'uncomfortable driving', for example, we use the following background knowledge B (only a fragment of B is shown here):

```
% LLE for bus b1
stop_enter(e1, init, pos1, 0, b1, bus, s8, scheduled)
stop_leave(e2, e1, pos1, 1, b1, bus, s8, late)
approach_intersection(e3, e2, pos1, 2, b1, bus)
abrupt_deceleration(e4, e3, pos1, 4, b1, bus)
sharp_turn(e5, e4, pos1, 6, b1, bus)
abrupt_acceleration(e6, e5, pos1, 8, b1, bus)
...

quantitative_distance(e2, e1, 1)
quantitative_distance(e3, e2, 1)
quantitative_distance(e4, e3, 2)
quantitative_distance(e5, e4, 2)
quantitative_distance(e6, e5, 2)
...

% domain knowledge
qualitative_distance(X, Y, very_long) ←
    quantitative_distance(X, Y, D),
    D>120

qualitative_distance(X, Y, long) ←
    quantitative_distance(X, Y, D),
    D>10, D<121

qualitative_distance(X, Y, short) ←
    quantitative_distance(X, Y, D),
    D>5, D<11

qualitative_distance(X, Y, very_short) ←
    quantitative_distance(X, Y, D),
    D<6
```

B includes the event narrative used for learning the hypotheses and a set of rules expressing domain

[5] http://www.doc.ic.ac.uk/~shm/Software/progol5.0/
[6] http://dtai.cs.kuleuven.be/ACE/
[7] http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/

knowledge. In this example, the narrative consists of a stream of detected CTM LLE. Each predicate representing a LLE, such as `approach_intersection`, has the following arguments: the first argument represents the `id` of the LLE, while the second argument represents the `id` of the directly temporally preceding LLE. `init` states that there is no temporally preceding LLE, that is, the LLE having `init` as the second argument is the first LLE of the sequence. The first two arguments of LLE specify event chaining in a symbolic manner. Such a symbolic representation of temporal relations between events is necessary because Aleph does not perform numerical reasoning. The third argument of a LLE predicate associates the LLE with a positive example from E^+ (such as the first positive example `pos1`) or a negative example from E^-, while the fourth argument represents the occurrence time of the LLE. The remaining arguments represent the standard properties of the LLE.

The `quantitative_distance` facts included in B express the temporal distance between two events. The `qualitative_distance` rules express domain knowledge: they translate the numerical temporal distances into symbolic distances (for example, `2` becomes `very_short`). This translation is necessary because, as mentioned earlier, Aleph does not perform numerical reasoning. The distance classification may be obtained by simple equal-frequency discretisation on a log of `quantitative_distance` facts, or by expertise if available.

Apart from the background knowledge B, we specify a set of mode declarations M — a type of language bias — to restrict the version space. Consider the following declarations specified for learning the definition of 'uncomfortable driving':

```
:- modeh(1,uncomfortable_driving(+ex))
:- modeb(1,stop_enter(-event,-event,+ex,-int,-id,-vehicletype,-stopid,-timetablecompliance))
:- modeb(1,stop_leave(-event,-event,+ex,-int,-id,-vehicletype,-stopid,-timetablecompliance))
:- modeb(1,approach_intersection(-event,-event,+ex,-int,-id,-vehicletype))
:- modeb(1,abrupt_deceleration(-event,-event,+ex,-int,-id,-vehicletype))
:- modeb(1,sharp_turn(-event,-event,+ex,-int,-id,-vehicletype))
:- modeb(1,abrupt_acceleration(-event,-event,+ex,-int,-id,-vehicletype))
:- modeb(*,qualitative_distance(+event,+event,#duration))
```

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 head or body of a hypothesis clause, and `r` is an integer, the 'recall', which limits how often the scheme is used. An asterisk `*` denotes an arbitrary recall. The placemarkers `+`, `-`, `#` express, respectively, input terms, output terms, and ground terms. Any input term in a body literal must be an input term in the head or an output term in some preceding body literal. A set M of mode declarations defines a language $\mathcal{L}(M) \subseteq \mathcal{L}_H$ within which a hypothesis H must fall, that is, $H \subseteq \mathcal{L}(M)$. A clause $\alpha \leftarrow l_1, \ldots, l_n$ belongs in $\mathcal{L}(M)$ if and only if the head literal α (respectively each body literal l_i) is obtained from some head (respectively body) declaration in M by replacing all # placemarkers with ground terms, all + placemarkers with input variables, and all − placemarkers with output variables. Given that $H \subseteq \mathcal{L}(M)$, the head of a hypothesis, in the presented illustration, is `uncomfortable_driving`, while in the body of a hypothesis we may have a predicate expressing the LLE shown above, as well as `qualitative_distance` expressing the temporal distance between these LLE. No other LLE may be in the body of `uncomfortable_driving` (we set this constraint due to prior knowledge about the subevents of `uncomfortable_driving`).

Other types of bias may also be used, such as setting the maximum number of body literals that a clause may contain.

Finally, to further guide the learning, we have set the following 'integrity constraint':

```
⊥ ← hypothesis(_, Body, _),
    broken_sequence(Body)
```

`hypothesis` is a built-in ALEPH predicate — the second argument of this predicate expresses the body of the hypothesis clause currently under consideration. `broken_sequence(Body)` is a user-defined predicate that is `true` when more than one literal (event) in the `Body` does not have its directly temporally preceding event in `Body`. The above integrity constraint, therefore, states that any candidate hypothesis clause in which there is more than one event in the body that does not have a directly temporally preceding event must be discarded. The absence of a temporally preceding event is allowed only for the initial event of a hypothesis clause.

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

```
% E+
uncomfortable_driving(pos1)
...

% E-
uncomfortable_driving(neg1)
...
```

Using the background knowledge B, various types of bias, integrity constraints, positive examples E^+ and negative examples E^-, ALEPH performs the following operations. First, it selects an example from E^+ to be generalised (such as `uncomfortable_driving(pos1)`). Second, it generates the most specific clause that entails this example with respect to B. In the case of `uncomfortable_driving(pos1)` the following clause is produced:

```
[bottom clause]
uncomfortable_driving(Ex) ←
    stop_enter(E1, Init, Ex, T1, Id, VehicleType, StopId, TC1),
    stop_leave(E2, E1, Ex, T2, Id, VehicleType, StopId, TC2),
    approach_intersection(E3, E2, Ex, T3, Id, VehicleType),
    abrupt_deceleration(E4, E3, Ex, T4, Id, VehicleType),
    sharp_turn(E5, E4, Ex, T5, Id, VehicleType),
    abrupt_acceleration(E6, E5, Ex, T6, Id, VehicleType),
    qualitative_distance(E2, E1, very_short),
    qualitative_distance(E3, E2, very_short),
    qualitative_distance(E4, E3, very_short),
    qualitative_distance(E5, E4, very_short),
    qualitative_distance(E6, E5, very_short)
```

Third, ALEPH searches for a more general clause than that generated in the previous step, aiming to cover as many positive examples from E^+ as possible, 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 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 case, comprises the following:

```
[Rule 1]
uncomfortable_driving(Ex) ←
    approach_intersection(E1, Init, Ex, T1, Id, VehicleType),
    abrupt_deceleration(E2, E1, Ex, T2, Id, VehicleType),
    sharp_turn(E3, E2, Ex, T3, Id, VehicleType),
    abrupt_acceleration(E4, E3, Ex, T4, Id, VehicleType),
    qualitative_distance(E2, E1, very_short),
    qualitative_distance(E3, E2, very_short),
    qualitative_distance(E4, E3, very_short)
```

The above clause is translated into the following CRS code:

```
chronicle uncomfortable_driving[?Id, ?VehicleType](T4) {
      event(approach_intersection[?Id, ?VehicleType], T1)
      event(abrupt_deceleration[?Id, ?VehicleType], T2)
      event(sharp_turn[?Id, ?VehicleType], T3)
      event(abrupt_acceleration[?Id, ?VehicleType], T4)
      T2 - T1 in [0, 5]
      T3 - T2 in [0, 5]
      T4 - T3 in [0, 5]
}
```

The above CRS code represents a less accurate account of the numerical temporal constraints than that of the definition presented in Section 2.2, but the HLE structure has been correctly discovered, without adding any unnecessary subevents such as `stop_leave`. Learning numerical temporal constraints with the use of ILP is an issue of current research.

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) was introduced by Kowalski and Sergot (1986) as a logic programming framework for representing and reasoning about events and their effects. Since then various alternative formalisations and implementations have been developed. As Miller and Shanahan (2002) point out, EC has been reformulated in various logic programming forms (for instance, (Sadri and Kowalski 1995; Chittaro and Montanari 1996; Denecker et al. 1996; Paschke 2005)), in classical logic (for example, (Shanahan 1999; Miller and Shanahan 1999; Cervesato et al. 2000; Mueller 2006a)), in modal logic (for example, (Cervesato et al. 1997; Cervesato et al. 1998; Cervesato et al. 2000)), and even in non-logic-based languages (such as (Farrell et al. 2005)).

EC has been frequently used for event recognition — to the best of our knowledge, only in a logic programming form, as in (Chittaro and Dojat 1997; Cervesato and Montanari 2000; Chaudet 2006; Paschke 2005; Paschke 2006; Paschke et al. 2007; Paschke and Bichler 2008; Teymourian and Paschke 2009; Artikis, Kukurikos et al. 2011; Artikis, Sergot and Paliouras 2010).

EC is related to other formalisms proposed in the literature of commonsense reasoning, such as the Situation Calculus (McCarthy and Hayes 1969; Reiter 2001), the action language $C+$ (Giunchiglia et al. 2004; Akman et al. 2004), the fluent calculus (Thielscher 1999; Thielscher 2001) and Temporal Action Logics (Doherty et al. 1998; Kvarnström 2005). Comparisons between formalisms for commonsense reasoning and proofs of equivalence between some of them may be found in (Kowalski and Sadri 1997; Miller

Table 3: Main Predicates of the Event Calculus.

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

and Shanahan 2002; Mueller 2006a; Mueller 2006b; Craven 2006), for example. The advantages of EC over other formalisms for commonsense reasoning with respect to event recognition are outlined in (Paschke 2005; Paschke and Kozlenkov 2009).

In the following section 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 and refining an EC logic program.

3.1 Representation

In this section we present features of typical EC dialects for event recognition (Artikis, Sergot and Paliouras 2010; Cervesato and Montanari 2000; Paschke 2005). 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, the effects of events, and the values of fluents. Table 3 presents typical predicates of EC dialects for event recognition. Variables start with an upper-case letter while predicates and constants start with a lower-case letter.

An EC dialect for event recognition has typically 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 (Miller and Shanahan 2002)). A partial specification of holdsAt, for example, is given below:

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

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

not represents 'negation by failure' (Clark 1978). 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 definitions of these two predicates are specific to the domain under consideration.

Alternatively, holdsAt may be defined in terms of holdsFor:

$$\begin{aligned} &\mathsf{holdsAt}(F=V,\ T) \leftarrow \\ &\quad \mathsf{holdsFor}(F=V,\ I), \\ &\quad (T_s, T_e) \in I, \\ &\quad T_s \leq T < T_e \end{aligned} \tag{3}$$

For any fluent F, $\mathsf{holdsAt}(F=V,\ T)$ if and only if time-point T belongs to one of the maximal intervals of I such that $\mathsf{holdsFor}(F=V, I)$. Intervals (T_s, T_e) in the presented EC representation correspond to $[T_s, T_e)$. Quite elaborate implementations of holdsFor have been proposed in the literature. In the following section we sketch an implementation of holdsFor. For alternative implementations the interested reader is referred to (Artikis, Sergot and Paliouras 2010; Cervesato and Montanari 2000).

To illustrate the use of EC for event recognition, below we present definitions of CTM HLE concerning vehicle (bus/tram) punctuality:

$$\begin{aligned} &\mathsf{happensAt}(\mathit{punctual}(\mathit{Id}, \mathit{VehicleType}),\ \mathit{DT}) \leftarrow \\ &\quad \mathsf{happensAt}(\mathit{stop\_enter}(\mathit{Id}, \mathit{VehicleType}, \mathit{StopId}, \mathit{scheduled}),\ \mathit{AT}), \\ &\quad \mathsf{happensAt}(\mathit{stop\_leave}(\mathit{Id}, \mathit{VehicleType}, \mathit{StopId}, \mathit{scheduled}),\ \mathit{DT}), \\ &\quad 1 \leq \mathit{DT} - \mathit{AT} \leq 2000 \end{aligned} \tag{4}$$

$$\begin{aligned} &\mathsf{happensAt}(\mathit{non\_punctual}(\mathit{Id}, \mathit{VehicleType}),\ \mathit{AT}) \leftarrow \\ &\quad \mathsf{happensAt}(\mathit{stop\_enter}(\mathit{Id}, \mathit{VehicleType}, \_, \mathit{late}),\ \mathit{AT}) \end{aligned} \tag{5}$$

According to the above formalisation, a vehicle is said to be punctual if it enters and leaves a stop at the scheduled time. The LLE 'stop enter' and 'stop leave' must take place within 2000 time-points in order to recognise 'punctual'. A vehicle is said to be non-punctual if it enters a stop later than the scheduled time. (These conditions are not the only ones in which a vehicle is said to be punctual/non-punctual.) 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:

$$\mathsf{initially}(\mathit{punctuality}(\_, \_) = \mathit{punctual}) \tag{6}$$

$$\begin{aligned} &\mathsf{initiatedAt}(\mathit{punctuality}(\mathit{Id}, \mathit{VehicleType}) = \mathit{punctual},\ T) \leftarrow \\ &\quad \mathsf{happensAt}(\mathit{punctual}(\mathit{Id}, \mathit{VehicleType}),\ T) \end{aligned} \tag{7}$$

$$\begin{aligned} &\mathsf{initiatedAt}(\mathit{punctuality}(\mathit{Id}, \mathit{VehicleType}) = \mathit{non\_punctual},\ T) \leftarrow \\ &\quad \mathsf{happensAt}(\mathit{non\_punctual}(\mathit{Id}, \mathit{VehicleType}),\ T) \end{aligned} \tag{8}$$

$$\begin{aligned} &\mathsf{happensAt}(\mathit{punctuality\_change}(\mathit{Id}, \mathit{VehicleType}, \mathit{Value}),\ T) \leftarrow \\ &\quad \mathsf{holdsFor}(\mathit{punctuality}(\mathit{Id}, \mathit{VehicleType}) = \mathit{Value},\ I), \\ &\quad (T, \_) \in I, \\ &\quad T \neq 0 \end{aligned} \tag{9}$$

We defined an auxiliary fluent, *punctuality*, that records the time-points in which a vehicle is (non-)punctual. The fluent *punctuality* is defined by rules (6)–(8). Initially, every vehicle is punctual. Thereafter *punctuality* is affected by the *punctual* and *non_punctual* HLE. Rule (9) expresses the definition of the HLE *punctuality_change*. This rule uses the EC built-in implementation of holdsFor to compute the list of 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 two lines of rule (9)).

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 Γ, for example, treating Γ as a fluent and using the built-in holdsFor to compute the intervals of Γ, may result in a more succinct representation than treating Γ as an EC event and developing domain-dependent rules for happensFor to compute the intervals of Γ.

Of interest to city transport officials are two HLE concerning driving quality — these are defined as follows:

$$\begin{aligned}&\mathsf{happensFor}(\mathit{medium\_quality\_driving}(\mathit{Id}, \mathit{VehicleType}),\ \mathit{MQDI}) \leftarrow \\ &\quad \mathsf{happensFor}(\mathit{uncomfortable\_driving}(\mathit{Id}, \mathit{VehicleType}),\ \mathit{UCI}), \\ &\quad \mathsf{holdsFor}(\mathit{punctuality}(\mathit{Id}, \mathit{VehicleType}) = \mathit{punctual},\ \mathit{PunctualI}), \\ &\quad \mathsf{intersect\_all}([\mathit{UCI}, \mathit{PunctualI}],\ \mathit{MQDI})\end{aligned} \tag{10}$$

$$\begin{aligned}&\mathsf{happensFor}(\mathit{low\_quality\_driving}(\mathit{Id}, \mathit{VehicleType}),\ \mathit{LQDI}) \leftarrow \\ &\quad \mathsf{happensFor}(\mathit{unsafe\_driving}(\mathit{Id}, \mathit{VehicleType}),\ \mathit{USI}), \\ &\quad \mathsf{holdsFor}(\mathit{punctuality}(\mathit{Id}, \mathit{VehicleType}) = \mathit{non\_punctual},\ \mathit{NPI}), \\ &\quad \mathsf{union\_all}([\mathit{USI}, \mathit{NPI}],\ \mathit{LQDI})\end{aligned} \tag{11}$$

happensFor represents the list of the maximal intervals during which an event takes place (see Table 3). Note that in the case where an event E has not taken place, we have $\mathsf{happensFor}(E,\ [])$. Similarly, if $F = V$ is never the case, we have $\mathsf{holdsFor}(F = V,\ [])$. intersect_all computes the intersection of a list of lists of maximal intervals. For example, $\mathsf{intersect\_all}([[(5, 20), (26, 30)], [(28, 35)]],\ [(28, 30)])$. Similarly, union_all computes the union of a list of lists of maximal intervals. For example, $\mathsf{union\_all}([[(5, 20), (26, 30)], [], [(28, 35)]], [(5, 20), (26, 35)])$. Medium quality driving is recognised when the driving style is uncomfortable, but the vehicle is punctual. Low quality driving is recognised when the driving style is unsafe or the vehicle is non-punctual. *punctuality* is defined by rules (6)–(8). To simplify the presentation we do not show here the definitions of the *uncofmortable_driving* and *unsafe_driving* HLE (see (Artikis, Kukurikos et al. 2011) for an extensive library of CTM HLE definitions formalised in EC).

The use of interval manipulation constructs, such as union_all and intersect_all, often leads to a very concise HLE representation. Cervesato and Montanari (2000) have used interval manipulation constructs in order to define a set of complex event operators in the context of EC. These operators are expressed as follows:

$$\begin{aligned} m\ ::=\ & e && \text{(basic event)} \\ |\ & m_1\ ;_d^D\ m_2 && \text{(sequence with delay } d \text{ to } D\text{)} \\ |\ & m_1\ +\ m_2 && \text{(alternative)} \\ |\ & m_1\ \|\ m_2 && \text{(parallelism)} \\ |\ & m^* && \text{(iteration)} \end{aligned}$$

The semantics of the above event operators is given in terms of a predicate that computes the interval of a complex event given the intervals of its sub-events.

In a similar way, Paschke (2005) has formalised the following event operators in the context of EC: sequence, disjunction, mutual exclusivity, conjunction, concurrency, negation, quantification and aperiodicity.

The availability of the full power of logic programming is one of the main attractions of employing EC as the temporal formalism. It allows HLE definitions to include not only complex temporal constraints, but also complex atemporal constraints. For example, it is straightforward to develop in Prolog a predicate computing the distance between two entities.

The cited EC dialects for event recognition are not tied to a particular type of logic programming semantics. In this way a dialect may be directly implemented in various existing rule languages (such as Prova).

Logic programming, not including an EC implementation, has been used frequently for event recognition. A notable example is the work of Shet and colleagues (Shet et al. 2005; Shet et al. 2006; Shet

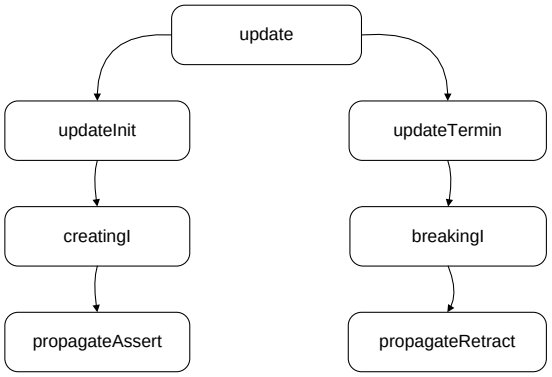

Fig. 6: The Cached Event Calculus.

et al. 2007). A benefit of EC, in comparison to pure Prolog, is that EC has built-in rules for complex temporal representation, such as the ones presented here, 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, and reasoning about the LLE log is performed when a query, concerning the recognition of HLE, is submitted. To perform run-time event recognition using query-time reasoning — to recognise, for example, at real-time incidents affecting the smooth operation of public transportation — 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, such as the recognition performed at the end of each day in order to evaluate the performance of public transportation, 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)* (Chittaro and Montanari 1996). 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. (Another EC dialect in which reasoning is performed at update-time may be found in (Chesani et al. 2009). We do not discuss here this dialect because its reasoning efficiency has not been evaluated yet.)

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

Figure 6 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. 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 break 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 order, the update has to be propagated to the fluents whose validity may rely on such

Table 4: Event Recognition in the Cached Event Calculus.

Input LLE	**Output HLE**
$\mathsf{happensAt}(stop\_enter(b_5, bus, s_8, late), 5)$	$\mathsf{happensAt}(non\_punctual(b_5, bus), 5)$ $\mathsf{holdsFor}(punctuality(b_5, bus) = non\_punctual, [(5, inf)])$ $\mathsf{holdsFor}(reducing\_passenger\_satisfaction(b_5, bus) = \mathsf{true}, [])$
$\mathsf{happensAt}(stop\_leave(b_5, bus, s_8, late), 12)$	$\mathsf{holdsFor}(punctuality(b_5, bus) = non\_punctual, [(5, inf)])$ $\mathsf{holdsFor}(reducing\_passenger\_satisfaction(b_5, bus) = \mathsf{true}, [])$
$\mathsf{happensAt}(passenger\_density\_change(b_5, bus, high), 15)$	$\mathsf{holdsFor}(punctuality(b_5, bus) = non\_punctual, [(5, inf)])$ $\mathsf{holdsFor}(reducing\_passenger\_satisfaction(b_5, bus) = \mathsf{true}, [(15, inf)])$
$\mathsf{happensAt}(stop\_enter(b_5, bus, s_9, scheduled), 13)$	$\mathsf{holdsFor}(punctuality(b_5, bus) = non\_punctual, [(5, inf)])$ $\mathsf{holdsFor}(reducing\_passenger\_satisfaction(b_5, bus) = \mathsf{true}, [(15, inf)])$
$\mathsf{happensAt}(stop\_leave(b_5, bus, s_9, scheduled), 14)$	$\mathsf{happensAt}(punctual(b_5, bus), 14)$ $\mathsf{holdsFor}(punctuality(b_5, bus) = non\_punctual, [(5, 14)])$ $\mathsf{holdsFor}(reducing\_passenger\_satisfaction(b_5, bus) = \mathsf{true}, [])$
...	

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 6, however, we do not show the information flow between *propagateAssert*, *propagateRetract* and the remaining CEC modules.

We will illustrate the way CEC deals with the non-chronological arrival of LLE with the use of a simple example. Assume that we are interested in recognising HLE related to vehicle punctuality (see rules (4)–(9)) and passenger satisfaction, a partial definition of which is presented below:

$$\begin{aligned}\mathsf{initiatedAt}(&reducing\_passenger\_satisfaction(Id, VehicleType) = \mathsf{true},\ T) \leftarrow \\ &\mathsf{happensAt}(passenger\_density\_change(Id, VehicleType, high),\ T), \\ &\mathsf{holdsAt}(punctuality(Id, VehicleType) = non\_punctual,\ T)\end{aligned} \tag{12}$$

According to the above formalisation, passenger satisfaction concerning a vehicle is said to be reducing when passenger density in that vehicle becomes high while the vehicle is non-punctual. Recall that, for any fluent F, $\mathsf{holdsAt}(F = V,\ T)$ if and only if time-point T belongs to one of the maximal intervals of I such that $\mathsf{holdsFor}(F = V,\ I)$ — see, for example, rule (3). Note that *reducing_passenger_satisfaction*, like *punctuality*, is represented as a fluent.

Table 4 shows an example LLE narrative concerning a particular vehicle, and a set of HLE that

are recognised given this narrative — more precisely, Table 4 shows the instantaneous *punctual* and *non_punctual* HLE recognised at each time, the maximal intervals for which a vehicle is said to be non-punctual, and the durative *reducing_passenger_satisfaction* HLE. A maximal interval (T_s, inf) indicates that a fluent holds continuously since T_s. Moreover, as mentioned earlier, intervals (T_s, T_e) in the presented EC representation correspond to $[T_s, T_e)$. Two LLE, in this example, arrive in a non-chronological order: $stop\_enter(b_5, bus, s_9, scheduled)$ and $stop\_leave(b_5, bus, s_9, scheduled)$. These LLE happened (were detected) before one of the already acquired LLE: $passenger\_density\_change(b_5, bus, high)$.

Upon the arrival of the first LLE of the narrative displayed in Table 4, CEC recognises that bus b_5 is non-punctual. This is due to the fact that b_5 entered stop s_8 later than the scheduled time. Passenger satisfaction is not affected at this time — the list of the intervals for which passenger satisfaction is reducing remains empty. The arrival of the second LLE does not lead to the recognition of a new HLE. b_5 is still considered non-punctual. The following LLE, $passenger\_density\_change(b_5, bus, high)$, leads to the recognition of $reducing\_passenger\_satisfaction(b_5, bus)$ because, at that time, bus b_5 is considered non-punctual (see rule (12)). The next two LLE arrive, as mentioned above, in a non-chronological order. They lead to the recognition of the $punctual(b_5, bus)$ HLE which is said to take place at time-point 14 (see rule (4)). This HLE breaks the interval for which b_5 is considered non-punctual. CEC, therefore, retracts the interval $(14, inf)$ for which $punctuality(b_5, bus) = non\_punctual$.

Given that the retraction of the aforementioned fluent interval was due to LLE that arrived in a non-chronological order, CEC has to propagate the retraction to the fluents whose validity relies on this interval — in this example, to the fluent representing the 'passenger satisfaction' HLE (see rule (12)). The retraction of the interval $(14, inf)$ for which $punctuality(b_5, bus) = non\_punctual$ modifies the context of the $passenger\_density\_change(b_5, bus, high)$ LLE that happened at time-point 15 — we now consider b_5 punctual at that time-point. CEC, therefore, invalidates the effects of $passenger\_density\_change(b_5, bus, high)$: it retracts the interval $(15, inf)$ for which $reducing\_passenger\_satisfaction(b_5, bus) = \mathsf{true}$.

The complexity of update processing (inferring and storing the consequences of an event) in CEC is measured in terms of accesses to happensAt and holdsFor Prolog facts, where happensAt facts represent the incoming LLE while holdsFor facts represent cached fluent intervals, including HLE intervals. On this basis, the complexity of update processing in CEC, considering a particular fluent, is $\mathcal{O}(n^{(L_{fw}+1)+2})$, where n is the number of initiating/terminating events for the fluent into consideration, and L_{fw} is the maximum number of propagations of fluent interval assertions and retractions — as shown above, such propagations are caused by LLE arriving in a non-chronological order. Note that if $L_{fw} = 0$ then the complexity of update processing is $\mathcal{O}(n^2)$. The complexity of query processing (retrieving the cached maximal intervals of a fluent) in CEC is $\mathcal{O}(n)$. Details about the complexity analysis of CEC may be found in (Chittaro and Montanari 1996).

The efficiency of CEC has been reported to be adequate for certain application domains (Chittaro and Dojat 1997). 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 (Artikis, Kukurikos et al. 2011). 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, *medium_quality_driving* — 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 (4) and (5). In general, learning hypotheses for predicates for which examples are available (such as happensAt($punctual(Id, VehicleType), T$)), that is, 'observation predicate learning' (OPL) (Muggleton and Bryant 2000), 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 — a description of abductive logic programming may be found at (Kakas et al. 1992; Denecker and Kakas 2000; Denecker and Kakas 2002), for example. 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 — see (Wellner 1999; Moyle 2002; Tamaddoni-Nezhad et al. 2006) for a few examples. In what follows we will briefly describe the XHAIL system (Ray 2009) 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 atoms 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 of XHAIL, including the inductive stage, is specified as an executable abductive logic programming task — see (Ray 2009) for details. In this way, all stages may be implemented using an abductive logic programming reasoner, such as the A-system[8], or a high-performance answer set solver, such as clasp[9].

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 definition includes:

- A background knowledge base B containing the built-in EC rules and a LLE narrative.
- A set M of mode declarations (the language bias). Mode declarations in XHAIL are specified as in typical ILP systems — see Section 2.3. Recall that a set M of mode declarations defines a language $\mathcal{L}(M)$ within which the learnt hypotheses H must fall, that is, $H \subseteq \mathcal{L}(M)$. We use M^+ to denote the set of head declarations in M and M^- to denote the set of body declarations. In the presented scenario, the head declaration in M^+ states that the head of learnt clause must be an initiatedAt predicate concerning the 'reducing passenger satisfaction' fluent, while the body declarations in M^- state that a body literal of a learnt clause may be a predicate representing *any* LLE, that is, we have no prior knowledge concerning what affects passenger satisfaction, or a predicate representing any fluent expressing in-vehicle conditions such as noise level, temperature and passenger density.

[8] http://dtai.cs.kuleuven.be/krr/Asystem/asystem.html
[9] http://www.cs.uni-potsdam.de/clasp/

- A set of positive and negative examples E such as:

$$\begin{aligned}&\mathsf{holdsAt}(\mathit{reducing\_passenger\_satisfaction}(b_1, \mathit{bus}) = \mathsf{true},\ \ 8)\\ &\mathsf{not\ holdsAt}(\mathit{reducing\_passenger\_satisfaction}(b_1, \mathit{bus}) = \mathsf{true},\ \ 16)\end{aligned}$$

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, and each α_i is a well-typed ground instance of a clause in $\mathcal{L}(M^+)$. This is a standard abductive task. Below is an atom produced by the abductive phase of XHAIL:

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

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

The second phase of XHAIL, that is, the deductive phase, computes a ground program $K = \bigcup_{i=1}^{n} \alpha_i \leftarrow \delta_i^1, \ldots, \delta_i^{m_i}$ 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, and each clause in K is a well-typed ground instance of a clause in $\mathcal{L}(M)$. n is the number of atoms abduced in the previous phase of XHAIL, while each m_i is less or equal to the number of body declarations. In other words, the second phase of XHAIL adds body literals to the clauses of the Kernel Set K. To compute K, each head atom computed in the previous phase is saturated with body literals using a non-monotonic generalisation of the Progol level saturation method (Muggleton 1995). To achieve this, the abductive system is made to behave as a deductive query answering procedure by setting an empty set of abducibles. Briefly, the atoms δ_i^j of each clause k_i of the Kernel Set are computed by a deductive procedure that finds the successful ground instances of the queries obtained by substituting a set of input terms into the + placemarkers of the body declaration schemas. Below is a clause of the produced Kernel Set K:

$$\begin{aligned}&\mathsf{initiatedAt}(\mathit{reducing\_passenger\_satisfaction}(b_1, \mathit{bus}) = \mathsf{true},\ \ 8) \leftarrow\\ &\quad \mathsf{happensAt}(\mathit{temperature\_change}(b_1, \mathit{bus}, \mathit{very\_warm}),\ \ 8),\\ &\quad \mathsf{holdsAt}(\mathit{punctuality}(b_1, \mathit{bus}) = \mathit{non\_punctual}),\ \ 8),\\ &\quad \mathsf{holdsAt}(\mathit{noise\_level}(b_1, \mathit{bus}) = \mathit{high}),\ \ 8)\end{aligned}$$

A temperature increase — more precisely, when the temperature becomes very warm — initiates a period of time for which passenger satisfaction is reducing, provided that the vehicle in question is non-punctual and the noise level inside the vehicle 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. Briefly, the Kernel Set K is translated into K' in which all input and output terms (recall that these are defined by means of mode declarations) are replaced by variables, and then as many literals and clauses as possible are deleted from K' while ensuring coverage of the examples in E. The resulting set of clauses constitutes H. Below is a clause of the computed theory H:

$$\begin{aligned}&\mathsf{initiatedAt}(\mathit{reducing\_passenger\_satisfaction}(\mathit{Id}, \mathit{VehicleType}) = \mathsf{true},\ \ T) \leftarrow\\ &\quad \mathsf{happensAt}(\mathit{temperature\_change}(\mathit{Id}, \mathit{VehicleType}, \mathit{very\_warm}),\ \ T),\\ &\quad \mathsf{holdsAt}(\mathit{punctuality}(\mathit{Id}, \mathit{VehicleType}) = \mathit{non\_punctual}),\ \ T)\end{aligned} \tag{13}$$

In-vehicle 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 above clause complements rule (12) that was presented in Section 3.2.

The proposed combination of abduction and induction has been applied to small, in terms of event narrative size, and noise-free applications (Ray 2009). 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 (Shet et al. 2007; Artikis, Sergot and Paliouras 2010): 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

Probabilistic graphical models are often used in the literature to handle uncertainty. Sequential graphical models, such as Dynamic Bayesian Networks (Murphy 2002) and Hidden Markov Models (Rabiner and Juang 1989) are useful for modelling HLE definitions representing event sequences. Reasoning with such models is usually performed through maximum likelihood estimation on the LLE narratives. For large-scale applications with complex events that involve long-range dependencies and hierarchical structure, sequential models have been extended to more complex variants (Hongeng and Nevatia 2003; Nguyen et al. 2005; Kersting et al. 2006). However, these models use a restricted temporal representation and most of them allow only for sequential relations between events. Moreover, the majority of them cannot 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 a first-order logic knowledge base expressing HLE definitions. A *possible world* assigns a truth value to each possible ground atom. Each formula in the knowledge base imposes constraints on the set of possible worlds. 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 that combine concepts from first-order logic and probabilistic models (Getoor and Taskar 2007; De Raedt and Kersting 2008; de Salvo Braz et al. 2008). 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 (de Salvo Braz et al. 2008). Markov Logic Networks (MLN) (Richardson and Domingos 2006; Domingos and Lowd 2009) is a recent and rapidly evolving KBMC framework, which provides a variety of reasoning and learning algorithms[10], and has recently been used for event recognition (Biswas et al. 2007; Tran and Davis 2008; Kembhavi et al. 2010; Xu and Petrou 2009; Helaoui et al. 2010; Wu and Aghajan 2011; Wu and Aghajan 2010). 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

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

'uncomfortable driving' CTM HLE:

$$\begin{aligned} &abrupt\_movement(Id, VehicleType, T) \leftarrow \\ &\quad abrupt\_acceleration(Id, VehicleType, T) \vee \\ &\quad abrupt\_deceleration(Id, VehicleType, T) \vee \\ &\quad sharp\_turn(Id, VehicleType, T) \end{aligned} \tag{14}$$

$$\begin{aligned} &uncomfortable\_driving(Id, VehicleType, T_2) \leftarrow \\ &\quad approach\_intersection(Id, VehicleType, T_1) \wedge \\ &\quad abrupt\_movement(Id, VehicleType, T_2) \wedge \\ &\quad before(T_1, T_2) \end{aligned} \tag{15}$$

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 reasoning techniques in the following section. According to the above formulas, *uncomfortable_driving* is defined in terms of an auxiliary construct, *abrupt_movement*, which is in turn defined in terms of the *abrupt_acceleration*, *abrupt_deceleration* and *sharp_turn* LLE. *before* is a simple predicate comparing two time-points. Formulas (14) and (15) are associated with 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 (15), 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', as normally a '*very* abrupt movement' is needed for 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 $C = c_1, ...c_{|C|}$. More precisely, all formulas are translated into *clausal form*, where the weight of each formula is equally divided among its clauses, and then the produced clauses are grounded. 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 Boolean *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, comprises nodes that correspond to 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 in X. A probability distribution over states is specified by the ground Markov network $M_{L,C}$, and represented as follows:

$$P(X = x) = \frac{1}{Z} exp\left(\sum_i^{|F_y|} w_i n_i(x)\right) \tag{16}$$

$F_y \subseteq F$ 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, $Z = \sum_{x \in \mathcal{X}} exp(\sum_i^{|F_y|} w_i n_i(x))$, where $\mathcal{X}$ is the set of all possible states.

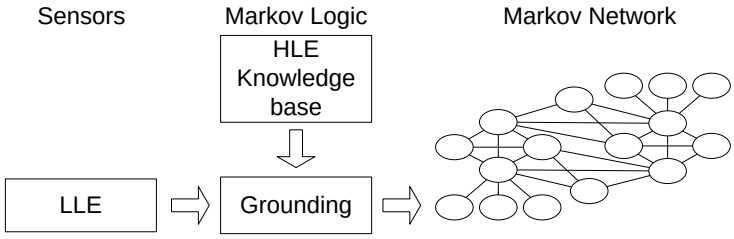

Fig. 7: Ground Markov Network Construction for Event Recognition.

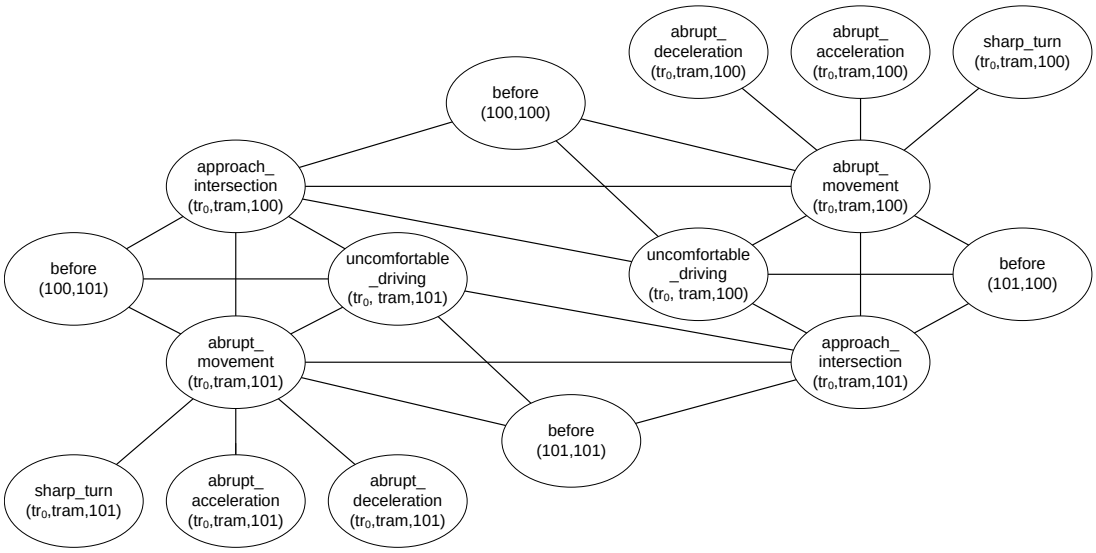

Fig. 8: Ground Markov Network.

We will illustrate the process of producing a ground Markov network and computing the probability of a state of such a network using the HLE definition expressed by formulas (14) and (15). These formulas are first translated into clausal form:

$$\frac{1}{3}w_1 \quad \neg abrupt\_acceleration(Id, VehicleType, T) \vee abrupt\_movement(Id, VehicleType, T) \tag{17}$$

$$\frac{1}{3}w_1 \quad \neg abrupt\_deceleration(Id, VehicleType, T) \vee abrupt\_movement(Id, VehicleType, T) \tag{18}$$

$$\frac{1}{3}w_1 \quad \neg sharp\_turn(Id, VehicleType, T) \vee abrupt\_movement(Id, VehicleType, T) \tag{19}$$

$$w_2 \quad \neg approach\_intersection(Id, VehicleType, T_1) \vee \neg abrupt\_movement(Id, VehicleType, T_2) \vee \neg before(T_1, T_2) \vee uncomfortable\_driving(Id, VehicleType, T_2) \tag{20}$$

w_1 is the weight of formula (14) — w_1 is equally divided among the three clauses of this formula — while w_2 is the weight of formula (15).

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 — see Figure 7. Consider, for example, a narrative of LLE about *tram* tr_0 taking place at time-points *100* and *101*. The constants involved in this narrative — $tr_0, tram, 100, 101$ — define the domain of the corresponding variables — Id, $VehicleType$, T, T_1, T_2. If more LLE were included in the input narrative then the variable domains would have been extended. The grounding procedure produces, in this example, 16 ground predicates

and 10 ground clauses. For example, clause (17) is grounded as follows:

$$\tfrac{1}{3}w_1 \quad \neg abrupt\_acceleration(tr_0, tram, 100) \ \vee abrupt\_movement(tr_0, tram, 100) \tag{17a}$$

$$\tfrac{1}{3}w_1 \quad \neg abrupt\_acceleration(tr_0, tram, 101) \ \vee abrupt\_movement(tr_0, tram, 101) \tag{17b}$$

Similarly, clause (20) is grounded as follows:

$$w_2 \quad \neg approach\_intersection(tr_0, tram, 100) \ \vee \neg abrupt\_movement(tr_0, tram, 100) \ \vee \neg before(100, 100) \vee uncomfortable\_driving(tr_0, tram, 100) \tag{20a}$$

$$w_2 \quad \neg approach\_intersection(tr_0, tram, 100) \ \vee \neg abrupt\_movement(tr_0, tram, 101) \ \vee \neg before(100, 101) \vee uncomfortable\_driving(tr_0, tram, 101) \tag{20b}$$

$$w_2 \quad \neg approach\_intersection(tr_0, tram, 101) \ \vee \neg abrupt\_movement(tr_0, tram, 100) \ \vee \neg before(101, 100) \vee uncomfortable\_driving(tr_0, tram, 100) \tag{20c}$$

$$w_2 \quad \neg approach\_intersection(tr_0, tram, 101) \ \vee \neg abrupt\_movement(tr_0, tram, 101) \ \vee \neg before(101, 101) \vee uncomfortable\_driving(tr_0, tram, 101) \tag{20d}$$

The resulting ground Markov network is shown in Figure 8. Predicates appearing in the same ground clause are connected to each other in the network and form a clique. Consider, for instance, the clique created by ground clause (17a) consisting of $abrupt\_acceleration(tr_0, tram, 100)$ and $abrupt\_movement(tr_0, tram, 100)$. The clique is associated with the weight of the corresponding clause, that is, $\frac{1}{3}w_1$, and a Boolean feature. In a state where $abrupt\_movement(tr_0, tram, 100)$ is true, the grounding (17a) of clause (17) is satisfied and therefore the value of the feature is 1.

We will illustrate how the probability of each state of a ground Markov network is computed by calculating the probability of two possible states, x_1 and x_2, of the network shown in Figure 8. Assume that both x_1 and x_2 assign the same truth values to all predicates except $uncomfortable\_driving(tr_0, tram, 101)$. More precisely, in both states $approach\_intersection(tr_0, tram, 100)$, $abrupt\_acceleration(tr_0, tram, 101)$, $abrupt\_movement(tr_0, tram, 101)$ and $before(100, 101)$ are true. All the other ground predicates are false, except $uncomfortable\_driving(tr_0, tram, 101)$ being true in x_1 and false in x_2. In this example, the weights w_1 and w_2 of formulas (14) and (15) are positive real numbers. The number of satisfied groundings of clauses (17), (18) and (19) is the same in both x_1 and x_2, as the assignment of truth values to the predicates involved is the same. The number of satisfied groundings of clause (20), however, differs between x_1 and x_2, because the truth assignment of $uncomfortable\_driving(tr_0, tram, 101)$ is different. As a result, in state x_1 all ground clauses of (20) are satisfied, but in state x_2 ground clause (20b) is not satisfied. Using equation (16) we compute the following:

$$P(X = x_1) = \tfrac{1}{Z} exp(\tfrac{1}{3}w_1 \cdot 2 + \tfrac{1}{3}w_1 \cdot 2 + \tfrac{1}{3}w_1 \cdot 2 + w_2 \cdot 4) = \tfrac{1}{Z} e^{2w_1 + 4w_2}$$

$$P(X = x_2) = \tfrac{1}{Z} exp(\tfrac{1}{3}w_1 \cdot 2 + \tfrac{1}{3}w_1 \cdot 2 + \tfrac{1}{3}w_1 \cdot 2 + w_2 \cdot 3) = \tfrac{1}{Z} e^{2w_1 + 3w_2}$$

According to the above results, a state (x_1) in which the 'uncomfortable driving' HLE and its subevents have all been recognised is more probable than a state (x_2) in which the subevents of 'uncomfortable driving' have been recognised while this HLE does not hold.

Event recognition in MLN involves querying a ground Markov network about HLE. 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', E is the set of 'evidence variables', and H is the set of the remaining variables with unknown value — also known as 'hidden variables'. In event recognition, query variables represent the HLE of interest, evidence variables represent the detected LLE, while hidden variables represent auxiliary constructs of a HLE definition. Event recognition queries require *conditional inference*, that is, computing the probability that a query variable holds given some evidence. For the computation of conditional probabilities a variety of exact and approximate probabilistic inference methods exist in the literature.

Given a MLN and some evidence $E = e$, a conditional query is specified as follows:

$$P(Q \mid E = e, H) = \frac{P(Q, E = e, H)}{P(E = e, H)} \tag{21}$$

Q are the query variables and H are the hidden variables. The numerator and denominator of this equation may be computed using equation (16). We may be interested in finding out, for instance, the trams that are driven in an uncomfortable manner, given a LLE narrative. In this case, the set of query variables Q includes only $uncomfortable\_driving(Id, VehicleType, T)$, the set of detected LLE that forms E may include, among others

$$approach\_intersection(tr_0, tram, 100)$$
$$abrupt\_acceleration(tr_0, tram, 101)$$
$$sharp\_turn(tr_{24}, tram, 100)$$

and the set of hidden variables H includes, among others

$$abrupt\_movement(tr_0, tram, 101)$$
$$abrupt\_movement(tr_{24}, tram, 101)$$

Given equation (21), we may compute the probability of each grounding of $uncomfortable\_driving(Id, VehicleType, T)$.

Complete grounding of MLN, even for simple HLE knowledge bases, results in complex and large networks. For this reason, only the minimal required network is constructed to answer a conditional query. In particular, query and 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, given the Markov network shown in Figure 8, we may be interested only in the HLE $uncomfortable\_driving(tr_0, tram, 101)$. The truth values of ground *before* predicates are trivially known. Moreover, the truth values of LLE are given as evidence — see Figure 9(a). The nodes for the ground predicates $abrupt\_movement(tr_0, tram, 100)$, $uncomfortable\_driving(tr_0, tram, 100)$, $sharp\_turn(tr_0, tram, 100)$, $abrupt\_deceleration(tr_0, tram, 100)$, $abrupt\_acceleration(tr_0, tram, 100)$, $before(100, 100)$ and $before(101, 100)$ can be omitted from the Markov network, as they cannot influence the reasoning process concerning the HLE $uncomfortable\_driving(tr_0, tram, 101)$. Therefore, the complete Markov network, shown in Figure 9(a), will be reduced into the network shown in Figure 9(b).

Further efficiency can be gained by employing (a) *lazy inference* methods that ground predicates as and when needed (Singla and Domingos 2006; Domingos and Lowd 2009, Section 3.3), or (b) *lifted inference* methods which can answer queries without grounding the entire network (Singla and Domingos 2008; Domingos and Lowd 2009, Section 3.4).

Sensors may detect LLE with certainty or with a degree of confidence. In the former case, the LLE are simple Boolean variables that are given directly to the MLN as evidence. In the latter case, the detected LLE are usually added to the MLN knowledge base as clauses having weights proportional to their detection probability (Tran and Davis 2008). For example, if the LLE $sharp\_turn(tr_0, tram, 20)$ is detected with some probability $P(sharp\_turn(tr_0, tram, 20))$, the unit clause $sharp\_turn(tr_0, tram, 20)$ will be added to the MLN with weight value

$$w = \log \frac{P(sharp\_turn(tr_0, tram, 20))}{1 - P(sharp\_turn(tr_0, tram, 20))}$$

The additional clauses, which represent the detected LLE with a degree of confidence, alter the posterior distribution over possible worlds by introducing additional features in the ground Markov Network. In this manner, the detected LLE propagate their degree of confidence to the whole MLN.

Even in a very large state space, the computation of equation (21) can be efficiently approximated by sampling methods, such as Markov Chain Monte Carlo (MCMC) algorithms — for example, Gibbs

Fig. 9: (a) Complete Ground Markov Network. (b) Ground Markov Network Reduced for Event Recognition. Nodes with a thick line represent query variables, shaded nodes represent variables with known truth values, and nodes with a dashed line represent variables which can be safely removed.

sampling. Markov Chains form graphs of possible states, over which a Monte Carlo simulation takes a random walk and draws a set of sample states from the target distribution. In MCMC, a successive sample state depends only on the current state. In Gibbs sampling, for instance, each sample state is produced by successively changing the truth value assignment of its ground predicates. This re-assignment of truth values is performed efficiently, as the value of each predicate in the ground Markov network depends only on the truth values of its neighbour predicates. The set of neighbouring predicates is called *Markov blanket* and represents predicates that appear together is some grounding of a clause. For instance, the Markov blanket of $abrupt\_movement(tr_0, tram, 101)$ in the network shown in Figure 8 is composed of $uncomfortable\_driving(tr_0, tram, 101)$, $before(100, 101)$, $approach\_intersection(tr_0, tram, 100)$, $sharp\_turn(tr_0, tram, 101)$, $abrupt\_acceleration(tr_0, tram, 101)$, $abrupt\_deceleration(tr_0, tram, 101)$, $approach\_intersection(tr_0, tram, 101)$ and $before(100, 101)$. The conditional probability in equation (21) can be computed by a MCMC algorithm that rejects all moves to states where $E = e$ does not hold. Therefore, only the variables that belong to Q and H are allowed to change in each sample. In each

step, the MCMC algorithm has the tendency to keep samples that represent states with high probability and, therefore, often converges in local maxima. The estimated probability of the query variables, that is, the HLE of interest, is the fraction of samples in which those variables are true. The more samples are generated, the more accurate this estimation becomes.

Due to the combination of logic with probabilistic models, inference in MLN must handle both deterministic and probabilistic dependencies. Deterministic or near-deterministic dependencies are formed from formulas with infinite and strong weights respectively. Being a purely statistical method, MCMC can only handle probabilistic dependencies. In the presence of deterministic dependencies, two important properties of Markov Chains, *ergodicity* and *detailed balance*, are violated and the sampling algorithms give poor results (Poon and Domingos 2006). Ergodicity is satisfied if all states are aperiodically reachable from each other, while detailed balance is satisfied if the probability of moving from state x_i to state x_j is the same as the probability of moving from x_j to x_i. Ergodicity and detailed balance are violated in the presence of deterministic dependencies because these dependencies create isolated regions in the state space by introducing zero-probability (impossible) states. Even near-deterministic dependencies create regions that are difficult to cross, that is, contain states with near zero-probability. As a result, typical MCMC methods, such as Gibbs sampling, get trapped in local regions. Thus, they are unsound for deterministic dependencies and they find it difficult to converge in the presence of near-deterministic ones.

To overcome these issues and deal with both deterministic and probabilistic dependencies, MLN use the MC-SAT algorithm (Poon and Domingos 2006; Domingos and Lowd 2009, Section 3.2), which is a MCMC method that combines satisfiability testing with *slice-sampling* (Damlen et al. 1999). Initially, a satisfiability solver is used to find those assignments that satisfy all hard-constrained clauses (that is, clauses with infinite weights). At each subsequent sampling step, MC-SAT chooses from the set of ground clauses satisfied by the current state the clauses that must be satisfied at the next step. Each clause is chosen with probability proportional to its weight value. Clauses with infinite or strong weights, that represent, respectively, deterministic and near-deterministic dependencies, will always be chosen with, respectively, absolute certainty and high probability. Then, instead of taking a sample from the space of all possible states, slice-sampling restricts sampling to the states that satisfy at least all chosen clauses. In this manner, MCMC cannot get trapped to local regions, as satisfiability testing helps to collect samples from all isolated and difficult-to-cross regions.

4.3 Machine Learning

Learning a MLN involves estimating the weights of the network and/or the first-order rules forming the network structure, given a set of training data, that is, LLE annotated with HLE. In Section 4.3.1 we present approaches on weight learning while in Section 4.3.2 we discuss structure learning.

4.3.1 Weight Learning

Weight learning concerns the estimation of the weights of the clauses that represent a HLE knowledge base — recall that the first-order rules of such a knowledge base are translated into clausal form. Different clauses, derived from the same rule, may be assigned different weights. For example, it may be estimated that the weights of the clauses derived from rule (14) are different:

$$w_a \quad \neg abrupt\_acceleration(Id, VehicleType, T) \vee abrupt\_movement(Id, VehicleType, T) \tag{22}$$

$$w_b \quad \neg abrupt\_deceleration(Id, VehicleType, T) \vee abrupt\_movement(Id, VehicleType, T) \tag{23}$$

$$w_c \quad \neg sharp\_turn(Id, VehicleType, T) \vee abrupt\_movement(Id, VehicleType, T) \tag{24}$$

Weight learning in MLN is performed by optimising a likelihood function, which is a statistical measure

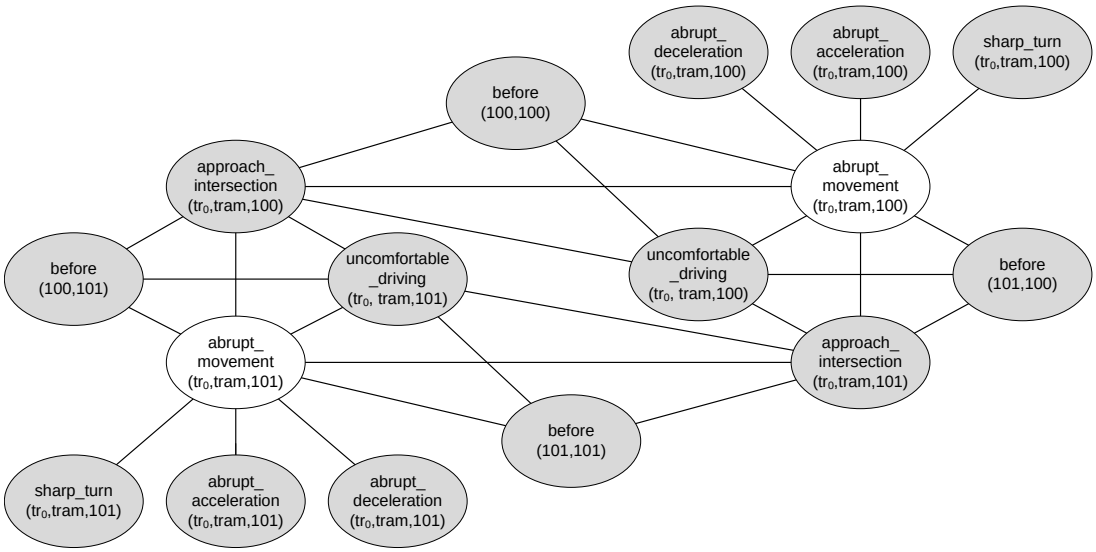

Fig. 10: Ground Markov Network Produced from Partially Annotated Dataset. The shaded nodes represent predicates with known truth values as described in the annotated dataset.

of how well the probabilistic model (MLN) fits the training data. In particular, weights can be learned by either generative or discriminative estimation (Singla and Domingos 2005; Lowd and Domingos 2007; Huynh and Mooney 2008; Domingos and Lowd 2009, 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-ascent optimisation algorithm. However, it has been shown that computing the likelihood and its gradient is intractable (Richardson and Domingos 2006). 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, that is, their Markov blanket. 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:

$$\log P_w^*(X = x) = \sum_{l=1}^{n} \log P_w(X_l = x_l \mid MB_x(X_l)) \tag{25}$$

$MB_x(X_l)$ 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 this assumption may lead to poor results. Consider, for example, formulas (14) and (15) expressing, respectively, *abrupt_movement* and *uncomfortable_driving*, and a training dataset including a LLE narrative annotated only for *uncomfortable_driving*, that is, there is no annotation for *abrupt_movement*. Figure 10 displays the resulting ground Markov network. As mentioned in Section 4.2, the Markov blanket of *abrupt_movement* includes the *sharp_turn*, *abrupt_acceleration* and *abrupt_deceleration* LLE. The Markov blanket of *uncomfortable_driving* includes *abrupt_movement*. In this case, pseudo-likelihood may give poor results with respect to the *uncomfortable_driving* HLE. This is due to the fact that it will only use information from the Markov blanket of this HLE, making an assumption about the absent annotation of *abrupt_movement* — usually a closed-world assumption. In other words, it will not exploit the

information provided by the *abrupt_acceleration*, *abrupt_deceleration* and *sharp_turn* LLE of the training dataset.

One way to address the aforementioned issue is to treat absent annotations as missing information and employ the expectation maximisation algorithm of (Dempster et al. 1977) in order to learn from incomplete data (Poon and Domingos 2008; Domingos and Lowd 2009, Section 4.3).

In event recognition, we know a-priori which variables form the evidence (LLE) and which ones concern queries (HLE). In the usual case, where we aim to recognise the latter given the former, it is preferable 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:

$$\log P(Q = q \mid E = e) = \sum_i w_i n_i(e, y) - \log Z_e \tag{26}$$

Z_e normalises over all states consistent with the evidence e, and $n_i(e, y)$ is the number of true groundings of the i-th clause in the training dataset. In contrast to Q and E, the state of a hidden variable, such as *abrupt_movement*, is unknown as, in most cases, the state of such variables is not available in the training dataset. In the presence of hidden variables, therefore, the conditional likelihood must be computed by marginalising over the hidden variables. It has been shown that learning weights discriminatively can lead to higher predictive accuracy than generative learning (Singla and Domingos 2005). 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.

Weights in MLN are estimated using either first-order or second-order optimisation methods. First-order methods, for example, perform gradient ascent optimisation in order to maximise an evaluation function, such as the conditional likelihood function (26). Weights may be initialised to zero, they may be given random values, or they may be given values according to other information concerning the application under consideration. Then, weights are iteratively updated according to the following equation:

$$\mathsf{w}_{t+1} = \mathsf{w}_t + \eta \mathsf{g} \tag{27}$$

t indicates the current iteration of the optimisation process, w_t is a vector that contains weight values of the current iteration for all clauses in the knowlege base, while w_{t+1} is a vector that contains the updated weight values. The constant η expresses the learning rate, defining the extent of weight adjustment in each iteration. The vector g represents the gradient. This is composed of the partial derivatives of function (26) with respect to the corresponding clause weights. The value of a derivative defines the direction and the magnitude of a weight update. The computation of the gradient requires inference, which can be efficiently approximated using MCMC methods, as discussed in Section 4.2.

By iteratively adjusting the weights using equation (27), the fit of the model to the HLE in the dataset is optimised. Assume, for example, that in a given training dataset the *abrupt_acceleration* LLE is rarely associated with the occurrence of the HLE *uncomfortable_driving*. On the other hand, the *abrupt_deceleration* and *sharp_turn* LLE are closely associated with the HLE *uncomfortable_driving*. In this case, weight learning will assign a low weight value to clause (22) and high weight values to clauses (23) and (24).

4.3.2 Structure Learning

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. In brief, these methods can be classified into top-down and bottom-up methods. Top-down structure learning (Kok and Domingos 2005; Domingos and Lowd 2009, 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. 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 (Mihalkova and Mooney 2007; Kok and Domingos 2009; Kok and Domingos 2010; Domingos and Lowd 2009, 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) — a LLE is created as a result of applying a computational derivation process to some other event. Using such input, event recognition systems identify high-level events (HLE) of interest, that is, collections of events that satisfy some pattern. We illustrated the use of the three reviewed approaches drawing examples from the domain of city transport management.

Being based on logic, all three approaches benefit from a formal and declarative semantics, a variety of inference mechanisms, and methods for learning a knowledge base of HLE definitions from data. As a result, compared to procedural methods, logic-based ones facilitate efficient development and management of HLE definitions, which are clearly separated from the generic inference mechanism. Moreover, compared to methods exhibiting informal semantics, logic-based approaches support validation and traceability of results. At the same time, 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 developers of CRS are currently making it open-source (C. Dousson, personal communication), thus facilitating extensions of its reasoning engine.

The Event Calculus (EC) provides a more generic and expressive representation of HLE definitions, 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) combine the strengths of logical and probabilistic inference. Consequently, they may address, to a certain extent, the issues of incomplete LLE streams, erroneous LLE detection, inconsistent LLE and HLE annotation, and limited dictionary of LLE and context variables. This is in contrast to CRS and EC that do not consider uncertainty in representation and reasoning. MLN also offer a very expressive framework for HLE definition representation, as the full power of first-order logic is available. The use of MLN for event recognition, however, has been 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 HLE definitions. Examples of such issues are the use of abduction to handle partial supervision in large datasets 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). Francois Portet is supported by the French national project Sweet-Home (ANR-09-VERS-011).

This paper is a significantly updated and extended version of (Artikis, Paliouras et al. 2010). We would like to thank the reviewers and participants of the Fourth International Conference on Distributed Event-Based Systems, as well as the reviewers of the Knowledge Engineering Review, who gave us useful feedback.

We should also like to thank Christophe Dousson for his suggestions regarding the Chronicle Recognition System. The authors themselves, however, are solely responsible for any misunderstanding about the use of this technology.

References

Akman, V., Erdogan, S., Lee, J., Lifschitz, V., and Turner, H. 2004. Representing the zoo world and the traffic world in the language of the Causal Calculator. *Artificial Intelligence 153,* 1–2, 105–140.

Allen, J. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM 26,* 11, 832–843.

Álvarez, M. R., Félix, P., Cariñena, P., and Otero, A. 2010. A data mining algorithm for inducing temporal constraint networks. In *International Conference on Information Processing and Management of Uncertainty (IPMU)*. 300–309.

Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal 15,* 2, 121–142.

Artikis, A., Kukurikos, A., Paliouras, G., Karampiperis, P., and Spyropoulos, C. 2011. Final version of knowledge base of event definitions and reasoning algorithms for event recognition. Deliverable 4.1.2 of EU-funded FP7 PRONTO project (FP7-ICT 231738). Available from the authors.

Artikis, A., Paliouras, G., Portet, F., and Skarlatidis, A. 2010. Logic-based representation, reasoning and machine learning for event recognition. In *Proceedings of Conference on Distributed Event-Based Systems (DEBS)*. ACM Press, 282–293.

Artikis, A., Sergot, M., and Paliouras, G. 2010. A logic programming approach to activity recognition. In *Proceedings of ACM Workshop on Events in Multimedia*.

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

Callens, L., Carrault, G., Cordier, M.-O., Fromont, É., Portet, F., and Quiniou, R. 2008. Intelligent adaptive monitoring for cardiac surveillance. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*. 653–657.

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

Cervesato, I., Franceschet, M., and Montanari, A. 1997. Modal event calculi with preconditions. In *Proceedings of Workshop on Temporal Reasoning (TIME)*. IEEE Computer Society, 38–45.

Cervesato, I., Franceschet, M., and Montanari, A. 1998. The complexity of model checking in modal event calculi with quantifiers. *Journal of Electronic Transactions on Artificial Intelligence*. `http://www.ida.liu.se/ext/etai`.

Cervesato, I., Franceschet, M., and Montanari, A. 2000. A guided tour through some extensions of the event calculus. *Computational Intelligence 16*, 307–347.

Cervesato, I. and Montanari, A. 2000. A calculus of macro-events: Progress report. In *Proceedings of the Seventh International Workshop on Temporal Representation and Reasoning (TIME)*. 47–58.

Chaudet, H. 2006. Extending the event calculus for tracking epidemic spread. *Artificial Intelligence in Medicine 38,* 2, 137–156.

Chesani, F., Mello, P., Montali, M., and Torroni, P. 2009. Commitment tracking via the reactive event calculus. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 91–96.

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

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

Choppy, C., Bertrand, O., and Carle, P. 2009. Coloured petri nets for chronicle recognition. In *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies*. Vol. LNCS 5570. Springer, 266–281.

Clark, K. 1978. Negation as failure. In *Logic and Databases*, H. Gallaire and J. Minker, Eds. Plenum Press, 293–322.

Craven, R. 2006. Execution mechanisms for the action language $\mathcal{C}+$. Ph.D. thesis, University of London.

Cugola, G. and Margara, A. 2011. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*.

Damlen, P., Wakefield, J., and Walker, S. 1999. Gibbs sampling for Bayesian non-conjugate and hierarchical models by using auxiliary variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 61,* 2, 331–344.

De Raedt, L. and Kersting, K. 2008. Probabilistic inductive logic programming. *Probabilistic inductive logic programming: theory and applications*, 1–27.

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

Dechter, R., Meiri, I., and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence 49*, 61–95.

Dempster, A., Laird, N., Rubin, D., et al. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological) 39,* 1, 1–38.

Denecker, M., Belleghem, K., Duchatelet, G., Piessens, F., and Schreye, D. 1996. A realistic experiment in knowledge representation in open event calculus: protocol specification. In *Proceedings of Joint International Conference and Symposium on Logic Programming (JICSLP)*, M. Maher, Ed. MIT Press, 170–184.

Denecker, M. and Kakas, A. 2000. Special issue: abductive logic programming. *Journal of Logic Programming 44,* 1-3, 1–4.

Denecker, M. and Kakas, A. 2002. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond*, A. Kakas and F. Sadri, Eds. Lecture Notes in Computer Science, vol. 2407. Springer, 99–134.

Doherty, P., Gustafsson, J., Karlsson, L., and Kvarnström, J. 1998. (TAL) temporal action logics: Language specification and tutorial. *Electronic Transactions on Artificial Intelligence 2,* 3–4, 273–306.

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

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

Dousson, C. 2002. Extending and unifying chronicle representation with event counters. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*. IOS Press, 257–261.

Dousson, C. and Duong, T. V. 1999. Discovering chronicles with numerical time constraints from alarm logs for monitoring dynamic systems. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 620–626.

Dousson, C., Gaborit, P., and Ghallab, M. 1993. Situation recognition: Representation and algorithms. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 166–174.

Dousson, C. and Maigat, P. L. 2006. Improvement of chronicle-based monitoring using temporal focalization and hierarchisation. In *Proceedings of International Workshop on Principles of Diagnosis (DX)*. 257–261.

Dousson, C. and Maigat, P. L. 2007. Chronicle recognition improvement using temporal focusing and hierarchisation. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 324–329.

Dousson, C., Pentikousis, K., Sutinen, T., and Mäkelä, J. 2007. Chronicle recognition for mobility management triggers. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*. 305–310.

Dzeroski, S. and Lavrac, N., Eds. 2001. *Relational Data Mining*. Springer.

Etzion, O. and Niblett, P. 2010. *Event Processing in Action*. Manning Publications Co.

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

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

Gao, F., Sripada, Y., Hunter, J., and Portet, F. 2009. Using temporal constraints to integrate signal analysis and domain knowledge in medical event detection. In *Artificial Intelligence in Medicine*. LNCS 5651. Springer, 46–55.

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

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

Ghallab, M. and Alaoui, A. M. 1989. Managing efficiently temporal relations through indexed spanning trees. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 1297–1303.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., and Turner, H. 2004. Nonmonotonic causal theories. *Artificial Intelligence 153,* 1–2, 49–104.

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

Helaoui, R., Niepert, M., and Stuckenschmidt, H. 2010. A statistical-relational activity recognition framework for ambient assisted living systems. In *ISAmI*, J. C. Augusto, J. M. Corchado, P. Novais, and C. Analide, Eds. Advances in Soft Computing, vol. 72. Springer, 247–254.

Hirate, Y. and Yamana, H. 2006. Sequential pattern mining with time intervals. In *Advances in Knowledge Discovery and Data Mining*. Springer, 775–779.

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

Huynh, T. and Mooney, R. 2008. Discriminative structure and parameter learning for Markov logic networks. In *Proceedings of the 25th international conference on Machine learning*. ACM, 416–423.

Kakas, A. C., Kowalski, R. A., and Toni, F. 1992. Abductive logic programming. *Journal of Logic and Computation 2,* 6, 719–770.

Kembhavi, A., Yeh, T., and Davis, L. S. 2010. Why did the person cross the road (there)? scene understanding using probabilistic logic models and common sense reasoning. In *ECCV (2)*, K. Daniilidis, P. Maragos, and N. Paragios, Eds. Lecture Notes in Computer Science, vol. 6312. Springer, 693–706.

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

Kohonen, T. 2001. *Self-Organising Maps*, 3rd ed. Springer.

Kok, S. and Domingos, P. 2005. Learning the structure of Markov logic networks. In *Proceedings of the 22nd international conference on Machine learning*. ACM, 441–448.

Kok, S. and Domingos, P. 2009. Learning Markov logic network structure via hypergraph lifting. In *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 505–512.

Kok, S. and Domingos, P. 2010. Learning markov logic networks using structural motifs. In *Proceedings of the International Conference on Machine Learning (ICML)*, J. Fürnkranz and T. Joachims, Eds. Omnipress, 551–558.

Konstantopoulos, S., Camacho, R., Fonseca, N., and Costa, V. S. 2008. Induction as a search. In *Artificial Intelligence for Advanced Problem Solving Techniques*, D. Vrakas and I. Vlahavas, Eds. IGI Global, Chapter VII, 158–205.

Kowalski, R. and Sadri, F. 1997. Reconciling the event calculus with the situation calculus. *Journal of Logic Programming 31*, 39–58.

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

Kvarnström, J. 2005. TALplanner and other extensions to temporal action logic. Ph.D. thesis, Department of Computer and Information Science, Linköping University.

Laer, W. V. 2002. From propositional to first order logic in machine learning and data mining. Ph.D. thesis, K. U. Leuven.

Le Guillou, X., Cordier, M.-O., Robin, S., and Rozé, L. 2008. Chronicles for on-line diagnosis of distributed systems. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*. 194–198.

Lowd, D. and Domingos, P. 2007. Efficient weight learning for Markov logic networks. *Knowledge Discovery in Databases: PKDD 2007*, 200–211.

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

Luckham, D. and Schulte, R. 2008. Event processing glossary — version 1.1. Event Processing Technical Society. `http://www.ep-ts.com/`.

Lv, F., Nevatia, R., and Lee, M. 2005. 3D human action recognition using spatio-temporal motion templates. In *Proceedings of International Workshop on Computer Vision in Human-Computer Interaction (ICCV)*. 120–130.

Mackworth, A. and Freuder, E. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence 25*, 65–74.

Mannila, H., Toivonen, H., and Verkamo, A. I. 1997. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov. 1,* 3, 259–289.

McCarthy, J. and Hayes, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4*, 463–502.

Mihalkova, L. and Mooney, R. 2007. Bottom-up learning of Markov logic network structure. In *Proceedings of the International Conference on Machine learning (ICML)*. ACM, 625–632.

Miller, R. and Shanahan, M. 1999. The event calculus in a classical logic — alternative axiomatizations. *Journal of Electronic Transactions on Artificial Intelligence 3,* A, 77–105.

Miller, R. and Shanahan, M. 2002. Some alternative formulations of the event calculus. In *Computational Logic: Logic Programming and Beyond — Essays in Honour of Robert A. Kowalski*. LNAI 2408. Spr, 452–490.

Morin, B. and Debar, H. 2003. Correlation of intrusion symptoms: an application of chronicles. In *6th International Conference on Recent Advances in Intrusion Detection (RAID'03)*. Pittsburgh, USA.

Moyle, S. 2002. Using theory completion to learn a robot navigation control program. In *Inductive Logic Programming*. Vol. LNCS 2583. Springer, 182–197.

Mueller, E. 2006a. *Commonsense Reasoning*. Morgan Kaufmann.

Mueller, E. 2006b. Event calculus and temporal action logics compared. *Artificial Intelligence 170,* 11, 1017–1029.

Muggleton, S. 1991. Inductive logic programming. *New Generation Computing 8,* 4, 295–318.

Muggleton, S. 1995. Inverse entailment and progol. *New Generation Computing 13,* 3–4, 245–286.

Muggleton, S. and Bryant, C. 2000. Theory completion using inverse entailment. In *Inductive Logic Programming*. Vol. LNCS 1866. Springer, 130–146.

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

Murphy, K. 2002. Dynamic bayesian networks: representation, inference and learning. Ph.D. thesis, University of California, Berkeley.

Nebel, B. and Bürckert, H.-J. 1995. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. *Journal of the ACM 42,* 1, 43–66.

Nédellec, C., Rouveirol, C., Adé, H., Bergadano, F., and Tausend, B. 1996. Declarative bias in ILP. In *Advances in Inductive Logic Programming*, L. D. Raedt, Ed. IOS Press, 82–103.

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

Paschke, A. 2005. ECA-RuleML: An approach combining ECA rules with temporal interval-based KR event/action logics and transactional update logics. Tech. Rep. 11, Technische Universität München.

Paschke, A. 2006. ECA-LP/ECA-RuleML: A homogeneous event-condition-action logic programming language. Tech. rep., CoRR abs/cs/0609143.

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

Paschke, A. and Kozlenkov, A. 2009. Rule-based event processing and reaction rules. In *Proceedings of RuleML*. Vol. LNCS 5858. Springer, 53–66.

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

Poon, H. and Domingos, P. 2006. Sound and efficient inference with probabilistic and deterministic dependencies. In *The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*. AAAI Press.

Poon, H. and Domingos, P. 2008. Joint unsupervised coreference resolution with Markov Logic. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 650–659.

Pottebaum, J. and Marterer, R. 2010. Final requirements, use case and scenario specification. In *Deliverable 6.1.2 of the EU-funded FP7 PRONTO project (FP7-ICT 231738)*. Available from the authors.

Quinlan, J. R. and Cameron-Jones, R. M. 1995. Induction of logic programs: Foil and related systems. *New Generation Computing 13*, 287–312.

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

Ray, O. 2009. Nonmonotonic abductive inductive learning. *Journal of Applied Logic 7,* 3, 329–340.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. The MIT Press.

Richardson, M. and Domingos, P. 2006. Markov logic networks. *Machine Learning 62,* 1-2, 107–136.

Sadri, F. and Kowalski, R. 1995. Variants of the event calculus. In *Proceedings of the International Conference on Logic Programming*. The MIT Press, 67–81.

Shanahan, M. 1999. The event calculus explained. In *Artificial Intelligence Today*, M. Wooldridge and M. Veloso, Eds. LNAI 1600. Springer, 409–430.

Shet, V., Harwood, D., and Davis, L. 2005. VidMAP: video monitoring of activity with Prolog. In *Proceedings of International Conference on Advanced Video and Signal Based Surveillance (AVSS)*. IEEE, 224–229.

Shet, V., Harwood, D., and Davis, L. 2006. Multivalued default logic for identity maintenance in visual surveillance. In *Proceedings of European Conference on Computer Vision (ECCV)*. LNCS 3954. Springer, 119–132.

Shet, V., Neumann, J., Ramesh, V., and Davis, L. 2007. Bilattice-based logical reasoning for human detection. In *Proceedings of International Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 1–8.

Singla, P. and Domingos, P. 2005. Discriminative training of Markov logic networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, M. M. Veloso and S. Kambhampati, Eds. 868–873.

Singla, P. and Domingos, P. 2006. Memory-efficient inference in relational domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Singla, P. and Domingos, P. 2008. Lifted first-order belief propagation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, D. Fox and C. P. Gomes, Eds. 1094–1099.

Tamaddoni-Nezhad, A., Chaleil, R., Kakas, A. C., and Muggleton, S. 2006. Application of abductive ILP to learning metabolic network inhibition from temporal data. *Machine Learning 64,* 1–3, 209–230.

Teymourian, K. and Paschke, A. 2009. Semantic rule-based complex event processing. In *Proceedings of RuleML*. Vol. LNCS 5858. Springer, 82–92.

Thielscher, M. 1999. From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence 111,* 1–2, 277–299.

Thielscher, M. 2001. The qualification problem: A solution to the problem of anomalous models. *Artificial Intelligence 131,* 1–2, 1–37.

Thonnat, M. 2008. Semantic activity recognition. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*. 3–7.

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

Vautier, A., Cordier, M.-O., and Quiniou, R. 2007. Towards data mining without information on knowledge structure. In *Knowledge Discovery in Databases*, J. Kok, J. Koronacki, R. Lopez de Mantaras, S. Matwin, D. Mladenic, and A. Skowron, Eds. 300–311.

Vilain, M. B. and Kautz, H. A. 1986. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 377–382.

Vu, V.-T., Brémond, F., and Thonnat, M. 2003. Automatic video interpretation: A novel algorithm for temporal scenario recognition. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 1295–1302.

Wellner, B. R. 1999. An abductive-inductive learning framework for logic-based agents. M.S. thesis, Imperial College of Science Technology and Medicine.

Wu, C. and Aghajan, H. K. 2010. Recognizing objects in smart homes based on human interaction. In *ACIVS (2)*, J. Blanc-Talon, D. Bone, W. Philips, D. C. Popescu, and P. Scheunders, Eds. Lecture Notes in Computer Science, vol. 6475. Springer, 131–142.

Wu, C. and Aghajan, H. K. 2011. User-centric environment discovery with camera networks in smart homes. *IEEE Transactions on Systems, Man, and Cybernetics, Part A 41,* 2, 375–383.

Xu, M. and Petrou, M. 2009. Learning logic rules for scene interpretation based on markov logic networks. In *ACCV (3)*, H. Zha, R. ichiro Taniguchi, and S. J. Maybank, Eds. Lecture Notes in Computer Science, vol. 5996. Springer, 341–350.

Yoshida, M., Iizuka, T., Shiohara, H., and Ishiguro, M. 2000. Mining sequential patterns including time intervals. In *Data Mining and Knowledge Discovery*. 213–220.