CrossMark

Incremental learning of event definitions with Inductive Logic Programming

Nikos Katzouris[1,2] · Alexander Artikis[1,3] ·
Georgios Paliouras[1]

Abstract Event recognition systems rely on knowledge bases of event definitions to infer occurrences of events in time. Using a logical framework for representing and reasoning about events offers direct connections to machine learning, via Inductive Logic Programming (ILP), thus allowing to avoid the tedious and error-prone task of manual knowledge construction. However, learning temporal logical formalisms, which are typically utilized by logic-based event recognition systems is a challenging task, which most ILP systems cannot fully undertake. In addition, event-based data is usually massive and collected at different times and under various circumstances. Ideally, systems that learn from temporal data should be able to operate in an incremental mode, that is, revise prior constructed knowledge in the face of new evidence. In this work we present an incremental method for learning and revising event-based knowledge, in the form of Event Calculus programs. The proposed algorithm relies on abductive–inductive learning and comprises a scalable clause refinement methodology, based on a compressive summarization of clause coverage in a stream of examples. We present an empirical evaluation of our approach on real and synthetic data from activity recognition and city transport applications.

✉ Nikos Katzouris
nkatz@iit.demokritos.gr

Alexander Artikis
a.artikis@iit.demokritos.gr

Georgios Paliouras
paliourg@iit.demokritos.gr

[1] Institute of Informatics and Telecommunications, National Center for Scientific Research "Demokritos", Athens, Greece

[2] Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Athens, Greece

[3] Department of Informatics, University of Piraeus, Piraeus, Greece

1 Introduction

The growing amounts of temporal data collected during the execution of various tasks within organizations are hard to utilize without the assistance of automated processes. Event recognition (Etzion and Niblett 2010; Luckham 2001; Luckham and Schulte 2008) refers to the automatic detection of event occurrences within a system. From a sequence of *low-level events* (for example sensor data) an event recognition system recognizes *high-level events* of interest, that is, events that satisfy some pattern. Event recognition systems with a logic-based representation of event definitions, such as the Event Calculus (Kowalski and Sergot 1986), are attracting significant attention in the event processing community for a number of reasons, including the expressiveness and understandability of the formalized knowledge, their declarative, formal semantics (Paschke 2005; Artikis et al. 2012) and their ability to handle rich background knowledge. Using logic programs in particular, has an extra advantage, due to the close connection between logic programming and machine learning in the field of Inductive Logic Programming (ILP) (Lavrač and Džeroski 1993; Muggleton and De Raedt 1994). However, such applications impose challenges that make most ILP systems inappropriate.

Several logical formalisms which incorporate time and change employ non-monotonic operators as a means for representing commonsense phenomena (Mueller 2006). Negation as Failure (NaF) is a prominent example. However, most ILP learners cannot handle NaF at all, or lack a robust NaF semantics (Sakama 2000; Ray 2009). Another problem that often arises when dealing with events, is the need to infer implicit or missing knowledge, for instance possible causes of observed events. In ILP the ability to reason with missing, or indirectly observable knowledge is called *non-Observational Predicate Learning (non-OPL)* (Muggleton 1995). This is a task that most ILP systems have difficulty to handle, especially when combined with NaF in the background knowledge (Ray 2006). One way to address this problem is through the combination of ILP with Abductive Logic Programming (ALP) (Denecker and Kakas 2002; Kakas and Mancarella 1990; Kakas et al. 1993). Abduction in logic programming is usually given a non-monotonic semantics (Eshghi and Kowalski 1989) and in addition, it is by nature an appropriate framework for reasoning with incomplete knowledge. The combination of ILP with ALP has a long history in the literature (Ade and Denecker 1995). However, only recently has it brought about systems such as XHAIL (Ray 2009), TAL (Corapi et al. 2010) and ASPAL (Corapi et al. 2012; Athakravi et al. 2013) that may be used for the induction of event-based knowledge.

The above three systems which, to the best of our knowledge, are the only ILP learners that address the aforementioned learnability issues, are *batch learners*, in the sense that all training data must be in place prior to the initiation of the learning process. This is not always suitable for event-oriented learning tasks, where data is often collected at different times and under various circumstances, or arrives in streams. In order to account for new training examples, a batch learner has no alternative but to re-learn a hypothesis from scratch. The cost is poor scalability when "learning in the large" (Dietterich et al. 2008) from a growing set of data. This is particularly true in the case of temporal data, which usually come in large volumes. Consider for instance data which span a large period of time, or sensor data transmitted at a very high frequency.

An alternative approach is learning incrementally, that is, processing training instances when they become available, and altering previously inferred knowledge to fit new observations. This process, also known as *Theory Revision* (Wrobel 1996), exploits previous computations to speed-up the learning, since revising a hypothesis is generally considered more efficient than learning it from scratch (Biba et al. 2006; Esposito et al. 2000; Cattafi et al. 2010). Numerous theory revision systems have been proposed in the literature, however their applicability in the presence of NaF is limited (Corapi et al. 2008). Additionally, as historical data grow over time, it becomes progressively harder to revise knowledge, so that it accounts both for new evidence and past experience. The development of scalable algorithms for theory revision has thus been identified as an important endeavour (Muggleton et al. 2012). One direction towards scaling theory revision systems is the development of techniques for reducing the need for reconsulting the whole history of accumulated experience, while updating existing knowledge.

This is the direction we take in this work. We build upon the ideas of non-monotonic ILP and use XHAIL as the basis for a scalable, incremental learner for the induction of event definitions in the form of Event Calculus theories. XHAIL has been used for the induction of action theories (Sloman and Lupu 2010; Alrajeh et al. 2010, 2011, 2012, 2009). Moreover, in Corapi et al. (2008) it has been used for theory revision in an incremental setting, revising hypotheses with respect to a recent, user-defined subset of the perceived experience. In contrast, the learner we present here performs revisions that account for all examples seen so far. We describe a compressive "memory" structure, which reduces the need for reconsulting past experience in response to a revision. Using this structure, we propose a method which, given a stream of examples, a theory which accounts for them and a new training instance, requires at most one pass over the examples in order to revise the initial theory, so that it accounts for both past and new evidence. We evaluate empirically our approach on real and synthetic data from an activity recognition application and a transport management application. Our results indicate that our approach is significantly more efficient than XHAIL, without compromising predictive accuracy, and scales adequately to large data volumes.

The rest of this paper is structured as follows. In Sect. 2 we present the Event Calculus dialect that we employ, describe the domain of activity recognition that we use as a running example and discuss abductive–inductive learning and XHAIL. In Sect. 3 we present our proposed method. In Sect. 4 we discuss some theoretical and practical implications of our approach. In Sect. 5 we present the experimental evaluation, and finally in Sects. 6 and 7 we discuss related work and draw our main conclusions.

2 Background

We assume a first-order language as in Lloyd (1987) where not in front of literals denotes NaF. We define the entailment relation between logic programs in terms of the *stable model semantics* (Gelfond and Lifschitz 1988)—see "Appendix 1" for details on the basics of logic programming used in this work. Following Prolog's convention, predicates and ground terms in logical formulae start with a lower case letter, while variable terms start with a capital letter.

2.1 The Event Calculus

The Event Calculus (Kowalski and Sergot 1986) is a temporal logic for reasoning about events and their effects. It is a formalism that has been successfully used in numerous event

Table 1 The basic predicates and axioms of SDEC

Predicate	Meaning
happensAt(E, T)	Event E occurs at time T
initiatedAt(F, T)	At time T a period of time for which fluent F holds is initiated
terminatedAt(F, T)	At time T a period of time for which fluent F holds is terminated
holdsAt(F, T)	Fluent F holds at time T
Axioms	

holdsAt($F, T + 1$) ←
    initiatedAt(F, T)
	holdsAt($F, T + 1$) ←
	    holdsAt(F, T),
	    not terminatedAt(F, T)

recognition applications (Paschke 2005; Artikis et al. 2015; Chaudet 2006; Cervesato and Montanari 2000). The ontology of the Event Calculus comprises *time points*, i.e. integers or real numbers; *fluents*, i.e. properties which have certain values in time; and *events*, i.e. occurrences in time that may affect fluents and alter their value. The domain-independent axioms of the formalism incorporate the common sense *law of inertia*, according to which fluents persist over time, unless they are affected by an event. We call the Event Calculus dialect used in this work Simplified Discrete Event Calculus (SDEC). It is a simplified version of the Discrete Event Calculus, a dialect which is equivalent to the classical Event Calculus when time ranges over integer domains (Mueller 2008).

The building blocks of SDEC and its domain-independent axioms are presented in Table 1. The first axiom in Table 1 states that a fluent F holds at time T if it has been initiated at the previous time point, while the second axiom states that F continues to hold unless it is terminated. initiatedAt/2 and terminatedAt/2 are defined in an application-specific manner.

Running example: activity recognition Throughout this paper we use the task of activity recognition, as defined in the CAVIAR[1] project, as a running example. The CAVIAR dataset consists of videos of a public space, where actors walk around, meet each other, browse information displays, fight and so on. These videos have been manually annotated by the CAVIAR team to provide the ground truth for two types of activity. The first type corresponds to low-level events, that is, knowledge about a person's activities at a certain time point (for instance *walking*, *running*, *standing still* and so on). The second type corresponds to high-level events, activities that involve more than one person, for instance two people *moving together*, *fighting*, *meeting* and so on. The aim is to recognize high-level events by means of combinations of low-level events and some additional domain knowledge, such as a person's position and direction at a certain time point.

Low-level events are represented in SDEC by streams of ground happensAt/2 atoms (see Table 2), while high-level events and other domain knowledge are represented by ground holdsAt/2 atoms. Streams of low-level events together with domain-specific knowledge will henceforth constitute the *narrative*, in ILP terminology, while knowledge about high-level events is the *annotation*. Table 2 presents an annotated stream of low-level events. We can see for instance that the person id_1 is *inactive* at time 999, her (x, y) coordinates are (201, 432) and her direction is 270°. The annotation for the same time point informs us that id_1 and id_2 are not moving together. Fluents express both high-level events and input information,

---

[1] http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1/.

Table 2 An annotated stream of low-level events

Narrative	Annotation
. . .	. . .
happensAt($inactive(id_1)$, 999)	not holdsAt($moving(id_1, id_2)$, 999)
happensAt($active(id_2)$, 999)	
holdsAt($coords(id_1, 201, 432)$, 999)	
holdsAt($coords(id_2, 230, 460)$, 999)	
holdsAt($direction(id_1, 270)$, 999)	
holdsAt($direction(id_2, 270)$, 999)	
happensAt($walking(id_1)$, 1000)	not holdsAt($moving(id_1, id_2)$, 1000)
happensAt($walking(id_2)$, 1000)	
holdsAt($coords(id_1, 201, 454)$, 1000)	
holdsAt($coords(id_2, 230, 440)$, 1000)	
holdsAt($direction(id_1, 270)$, 1000)	
holdsAt($direction(id_2, 270)$, 1000)	
happensAt($walking(id_1)$, 1001)	holdsAt($moving(id_1, id_2)$, 1001)
happensAt($walking(id_2)$, 1001)	
holdsAt($coords(id_1, 201, 454)$, 1001)	
holdsAt($coords(id_2, 227, 440)$, 1001)	
holdsAt($direction(id_1, 275)$, 1001)	
holdsAt($direction(id_2, 278)$, 1001)	
. . .	. . .

such as the coordinates of a person. We discriminate between *inertial* and *statically defined* fluents. The former should be inferred by the Event Calculus axioms, while the latter are provided with the input.

2.2 Abductive–inductive learning

Given a domain description in the language of SDEC, the aim of machine learning addressed in this work is to derive the *Domain-Specific Axioms*, that is, the axioms that specify how the occurrence of low-level events affect the truth values of fluents that represent high-level events, by initiating or terminating them. Thus, we wish to learn initiatedAt/2 and terminatedAt/2 definitions from positive and negative examples.

Inductive Logic Programming (ILP) refers to a set of techniques for learning hypotheses in the form of clausal theories, and is the machine learning approach that we adopt in this work. An ILP task is a triplet $ILP(B, E, M)$ where B is some background knowledge, M is some language bias and $E = E^+ \cup E^-$ is a set of positive (E^+) and negative (E^-) examples for the target predicates, represented as logical facts. The goal is to derive a set of non-ground clauses H in the language of M that *cover the examples* w.r.t. B, i.e. $B \cup H \models E^+$ and $B \cup H \nvDash E^-$.

Henceforth, the term "example" encompasses anything known true at a specific time point. We assume a closed world, thus anything that is not explicitly given is considered false (to avoid confusion, in the tables throughout the paper we state both positive and negated annotation atoms). An example is either positive or negative based on the annotation. For

instance the examples at times 999 and 1000 in Table 2 are negative, while the example at time 1001 is positive.

Learning event definitions in the form of domain-specific Event Calculus axioms with ILP requires non-Observational Predicate Learning (non-OPL) (Muggleton 1995), meaning that instances of target predicates (initiatedAt/2 and terminatedAt/2) are not provided with the supervision, which consists of holdsAt/2 atoms. A solution is to use Abductive Logic Programming (ALP) to obtain the missing instances. An ALP task is a triplet $ALP(B, A, G)$ where B is some background knowledge, G is a set of observations that must be explained, represented by ground logical atoms, and A is a set of predicates called *abducibles*. An explanation Δ is a set of ground atoms from A such that $B \cup \Delta \vDash G$.

2.2.1 The XHAIL system

XHAIL is an abductive–inductive system that constructs hypotheses in a three-phase process. Given an ILP task $ILP(B, E, M)$, the first two phases return a ground program K, called *Kernel Set of E*, such that $B \cup K \vDash E$. The first phase generates the heads of K's clauses by abductively deriving from B a set Δ of instances of head atoms, as defined by the language bias, such that $B \cup \Delta \vDash E$. The second phase generates K, by saturating each previously abduced atom with instances of body atoms that deductively follow from $B \cup \Delta$. The language bias used by XHAIL is *mode declarations* (see "Appendix 1" for a formal account).

By construction, the Kernel Set covers the provided examples. In order to find a good hypothesis, XHAIL thus searches in the space of theories that subsume the Kernel Set. To this end, the latter is variabilized, i.e. each term that corresponds to a variable, according to the language bias, is replaced by an actual variable. The variablized Kernel Set K_v is subject to a syntactic transformation of its clauses, which involves two new predicates *try/3* and *use/2*.

For each clause $C_i \in K_v$ and each body literal $\delta_i^j \in C_i$, a new atom $v(\delta_i^j)$ is generated, as a special term that contains the variables that appear in δ_i^j. The new atom is wrapped inside an atom of the form $try(i, j, v(\delta_i^j))$. An extra atom $use(i, 0)$ is added to the body of C_i and two new clauses $try(i, j, v(\delta_i^j)) \leftarrow use(i, j), \delta_i^j$ and $try(i, j, v(\delta_i^j)) \leftarrow$ not $use(i, j)$ are generated, for each body literal $\delta_i^j \in C_i$.

All these clauses are put together into a program U_{K_v}. U_{K_v} serves as a "defeasible" version of K_v from which literals and clauses may be selected in order to construct a hypothesis that accounts for the examples. This is realized by solving an ALP task with *use/2* as the only abducible predicate. As explained in Ray (2009), the intuition is as follows: In order for the head atom of clause $C_i \in U_{K_v}$ to contribute towards the coverage of an example, each of its $try(i, j, v(\delta_i^j))$ atoms must succeed. By means of the two rules added for each such atom, this can be achieved in two ways: Either by assuming not $use(i, j)$, or by satisfying δ_i^j and abducing $use(i, j)$. A hypothesis clause is constructed by the head atom of the i-th clause C_i of K_v, if $use(i, 0)$ is abduced, and the j-th body literal of C_i, for each abduced $use(i, j)$ atom. All other clauses and literals from K_v are discarded. Search is biased by minimality, i.e. preference towards hypotheses with fewer literals. This is realized by means of abducing a minimal set of $use/2$ atoms.

Example 1 Table 3 presents the process of hypothesis generation by XHAIL. The input consists of a set of examples, a set of mode declarations (omitted for simplicity) and the axioms of SDEC as background knowledge. The annotation says that *fighting* between persons id_1 and id_2 holds at time 1 and it does not hold at times 2 and 3, hence it is terminated

Table 3 Hypothesis generation by XHAIL for Example 1

Input

Narrative	**Annotation**
happensAt(*abrupt(id₁)*, *1*).	holdsAt(*fighting(id₁, id₂)*, *1*).

happensAt(*abrupt(id$_1$)*, *1*).   holdsAt(*fighting(id$_1$, id$_2$)*, *1*).

happensAt(*walking(id$_2$)*, *1*).   not holdsAt(*fighting(id$_3$, id$_4$)*, *1*).

not holdsAt(*close(id$_1$, id$_2$, 23)*, *1*).   not holdsAt(*fighting(id$_1$, id$_2$)*, *2*).

happensAt(*abrupt(id$_3$)*, *2*).   not holdsAt(*fighting(id$_3$, id$_4$)*, *2*).

happensAt(*abrupt(id$_4$)*, *2*).   not holdsAt(*fighting(id$_1$, id$_2$)*, *3*).

holdsAt(*close(id$_3$, id$_4$, 23)*, *2*).   holdsAt(*fighting(id$_3$, id$_4$)*, *3*).

Mode declarations   **Background knowledge:** SDEC (Table 1)

Phase 1 (Abduction):

Δ_1={initiatedAt(*fighting(id$_3$, id$_4$)*, *2*),

terminatedAt(*fighting(id$_1$, id$_2$)*, *1*)}

Phase 2 (Deduction):

Kernel Set K:	**Variabilized Kernel Set K_v:**

initiatedAt(*fighting(id$_3$, id$_4$)*, *2*) $\leftarrow$ initiatedAt(*fighting(X, Y)*, *T*) $\leftarrow$

  happensAt(*abrupt(id$_3$)*, *2*),   happensAt(*abrupt(X)*, *T*),

  happensAt(*abrupt(id$_4$)*, *2*),   happensAt(*abrupt(Y)*, *T*),

  holdsAt(*close(id$_3$, id$_4$, 23)*, *2*).   holdsAt(*close(X, Y, 23)*, *T*).

terminatedAt(*fighting(id$_1$, id$_2$)*, *1*) $\leftarrow$ terminatedAt(*fighting(X, Y)*, *T*) $\leftarrow$

  happensAt(*abrupt(id$_1$)*, *1*),   happensAt(*abrupt(X)*, *T*),

  happensAt(*walking(id$_2$)*, *1*),   happensAt(*walking(Y)*, *T*),

  not holdsAt(*close(id$_1$, id$_2$, 23)*, *1*).   not holdsAt(*close(X, Y, 23)*, *T*).

Phase 3 (Induction):

Program U_{K_v} (Syntactic transformation of K_v):

initiatedAt(*fighting(X, Y)*, *T*) $\leftarrow$ terminatedAt(*fighting(X, Y)*, *T*) $\leftarrow$

  use(1, 0), *try(1, 1, vars(X, T))*,   *use(2, 0)*, *try(2, 1, vars(X, T))*,

  try(1, 2, vars(Y, T)),   *try(2, 2, vars(Y, T))*,

  try(1, 3, vars(X, Y, T)).   *try(2, 3, vars(X, Y, T))*.

try(1, 1, vars(X, T)) $\leftarrow$ *try(2, 1, vars(X, T))* $\leftarrow$

  use(1, 1), happensAt(*abrupt(X)*, *T*).   *use(2, 1)*, happensAt(*abrupt(X)*, *T*).

try(1, 1, vars(X, T)) $\leftarrow$ not *use(1, 1)*.   *try(2, 1, vars(X, T))* $\leftarrow$ not *use(1, 1)*.

try(1, 2, vars(Y, T)) $\leftarrow$ *try(2, 2, vars(Y, T))* $\leftarrow$

  use(1, 2), happensAt(*abrupt(Y)*, *T*).   *use(2, 2)*, happensAt(*walking(Y)*, *T*).

try(1, 2, vars(X, T)) $\leftarrow$ not *use(1, 2)*.   *try(2, 2, vars(Y, T))* $\leftarrow$ not *use(2, 2)*.

try(1, 3, vars(X, Y, T)) $\leftarrow$ *try(2, 3, vars(X, Y, T))* $\leftarrow$

  use(1, 3), holdsAt(*close(X, Y, 23)*, *T*).   *use(2, 3)*, not holdsAt(*close(X, Y, 23)*, *T*).

try(1, 3, vars(X, T)) $\leftarrow$ not *use(1, 3)*.   *try(2, 3, vars(X, Y, T))* $\leftarrow$ not *use(2, 3)*.

Search:	**Abductive Solution:**

ALP(SDEC $\cup$ U_{K_v}, {*use/2*}, *Narrative $\cup$ Annotation*) Δ_2 = {*use(1, 0)*, *use(1, 3)*,

       use(2, 0), *use(2, 2)*}

Output hypothesis

initiatedAt(*fighting(X, Y)*, *T*) $\leftarrow$ terminatedAt(*fighting(X, Y)*, *T*) $\leftarrow$

  holdsAt(*close(X, Y, 23)*, *T*).   happensAt(*walking(Y)*, *T*).

---

at time 1. Respectively, *fighting* between persons id_3 and id_4 holds at time 3 and does not hold at times 1 and 2, hence it is initiated at time 2. XHAIL obtains these explanations for the holdsAt/2 literals of the annotation abductively, using the head mode declarations as abducibles. In its first phase, it derives the two ground atoms in Δ_1 (Phase 1, Table 3). In its second phase, XHAIL forms a Kernel Set (Phase 2, Table 3), by generating one clause from each abduced atom in Δ_1, using this atom as head, and body literals that deductively follow from SDEC $\cup$ Δ_1 as the body of the clause.

The Kernel Set is variabilized and the third phase of XHAIL functionality concerns the actual search for a hypothesis. This search is biased by *minimality*, i.e. preference towards hypotheses with fewer literals. A hypothesis is thus constructed by dropping as many literals and clauses from K_v as possible, while correctly accounting for all the examples. The syntactic transformation on K_v (see also Sect. 2.2.1) results in the defeasible program U_{K_v}.

Literals and clauses necessary to cover the examples are selected from U_{K_v} by means of abducing a set of *use*/2 atoms, as explanations of the examples, from the ALP task presented in Phase 3 of Table 3. Δ_2 from Table 3 is a minimal explanation for this ALP task. *use*$(1, 0)$ and *use*$(2, 0)$ correspond to the head atoms of the two K_v clauses, while *use*$(1, 3)$ and *use*$(2, 2)$ correspond respectively to their third and second body literal. The output hypothesis in Table 3 is constructed by these literals, while all other literals and clauses from K_v are discarded.   □

XHAIL provides an appropriate framework for learning Event Calculus programs. A major drawback however is that it scales poorly, partly because of the increased computational complexity of adbuction, which lies at the core of its functionality, and partly because of the combinatorial complexity of learning whole theories, which may result in an intractable search space. In what follows, we use the XHAIL machinery to develop a novel incremental algorithm that scales to large volumes of sequential data, typical of event-based applications.

3 ILED: incremental learning of event definitions

A hypothesis H is called *incomplete* if it does not account for some positive examples and *inconsistent* if it erroneously accounts for some negative examples. Incompleteness is typically treated by *generalization*, e.g. addition of new clauses, or removal of literals from existing clauses. Inconsistency is treated by *specialization*, e.g. removal of clauses, or addition of new literals to existing clauses. *Theory revision* is the process of acting upon a hypothesis, in order to change the examples it accounts for. Theory revision is at the core of incremental learning settings, where examples are provided over time. A learner induces a hypothesis from scratch, from the first available set of examples, and treats this hypothesis as a revisable background theory in order to account for new examples.

Definition 1 provides a concrete account of the incremental setting we assume for our approach, which we call incremental learning of event definitions (ILED).

Definition 1 (*Incremental learning*) We assume an ILP task ILP(SDEC, $\mathcal{E}$, M), where $\mathcal{E}$ is a database of examples, called historical memory, storing examples presented over time. Initially $\mathcal{E} = \emptyset$. At time n the learner is presented with a hypothesis H_n such that SDEC$\cup H_n \vDash$ $\mathcal{E}$, in addition to a new set of examples w_n. The goal is to revise H_n to a hypothesis H_{n+1}, so that SDEC $\cup$ $H_{n+1} \vDash \mathcal{E} \cup w_n$.

A main challenge of adopting a full memory approach is to scale it up to a growing size of experience. This is in line with a key requirement of incremental learning where "the

incorporation of experience into memory during learning should be computationally efficient, that is, theory revision must be efficient in fitting new incoming observations" (Langley 1995; Di Mauro et al. 2005). In the stream processing literature, the number of passes over a stream of data is often used as a measure of the efficiency of algorithms (Li et al. 2004; Li and Lee 2009). In this spirit, the main contribution of ILED, in addition to scaling up XHAIL, is that it adopts a "single-pass" theory revision strategy, that is, a strategy that requires at most one pass over $\mathcal{E}$ in order to compute H_{n+1} from H_n.

Since experience may grow over time to an extent that is impossible to maintain in the working memory, we follow an external memory approach (Biba et al. 2006). This implies that the learner does not have access to all past experience as a whole, but to independent sets of training data, in the form of *sliding windows*. At time n, ILED is presented with a hypothesis H_n that accounts for the historical memory so far, and a new example window w_n. If H_n covers the new window then it is returned as is, otherwise ILED starts the process of revising H_n. In this process, revision operators that retract knowledge, such as the deletion of clauses or antecedents are excluded, due to the exponential cost of backtracking in the historical memory (Badea 2001). The supported revision operators are thus:

– Addition of new clauses.
– Refinement of existing clauses, i.e. replacement of an existing clause with one or more specializations of that clause.

To treat incompleteness we add initiatedAt clauses and refine terminatedAt clauses, while to treat inconsistency we add terminatedAt clauses and refine initiatedAt clauses. The goal is to retain the *preservable* clauses of H_n intact, refine its *revisable* clauses and, if necessary, generate a set of new clauses that account for new examples in the incoming window w_n. We henceforth call a clause preservable w.r.t. a set of examples if it does not cover negatives, nor it disproves positives, and call it revisable otherwise.

Figure 1 illustrates the revision process with a simple example. New clauses are generated by generalizing a Kernel Set of the incoming window, as shown in Fig. 1, where a terminatedAt/2 clause is generated from the new window w_n. To facilitate refinement of existing clauses, each clause in the running hypothesis is associated with a memory of the examples it covers throughout $\mathcal{E}$, in the form of a "bottom program", which we call *support set*. The support set is constructed gradually, from previous Kernel Sets, as new example windows arrive. It serves as a refinement search space, where the single clause in the running hypothesis H_n is refined w.r.t. the incoming window w_n into two specializations. Each such specialization is constructed by adding to the initial clause one antecedent from the two support set clauses which are presented in Fig. 1. The revised hypothesis H_{n+1} is constructed from the refined clauses and the new ones, along with the preserved clauses of H_n, if any.

ILED's support set can be seen as the S-set in a version space (Mitchell 1979), i.e. the space of all overly-specific hypotheses, progressively augmented while new examples arrive. Similarly, a running hypothesis of ILED can be seen as an element of the G-set in a version space, i.e. the space of all overly-general hypotheses that account for all examples seen so far, and need to be further refined as new examples arrive.

There are two key features of ILED that contribute towards its scalability: First, re-processing of past experience is necessary only in the case where new clauses are generated by a revision, and is redundant in the case where a revision consists of refinements of existing clauses only. Second, re-processing of past experience requires a single pass over the historical memory, meaning that it suffices to re-visit each past window exactly once to ensure that the output revised hypothesis H_{n+1} is complete & consistent w.r.t. the entire historical memory. These properties of ILED are due to the support set, which we next present in detail.
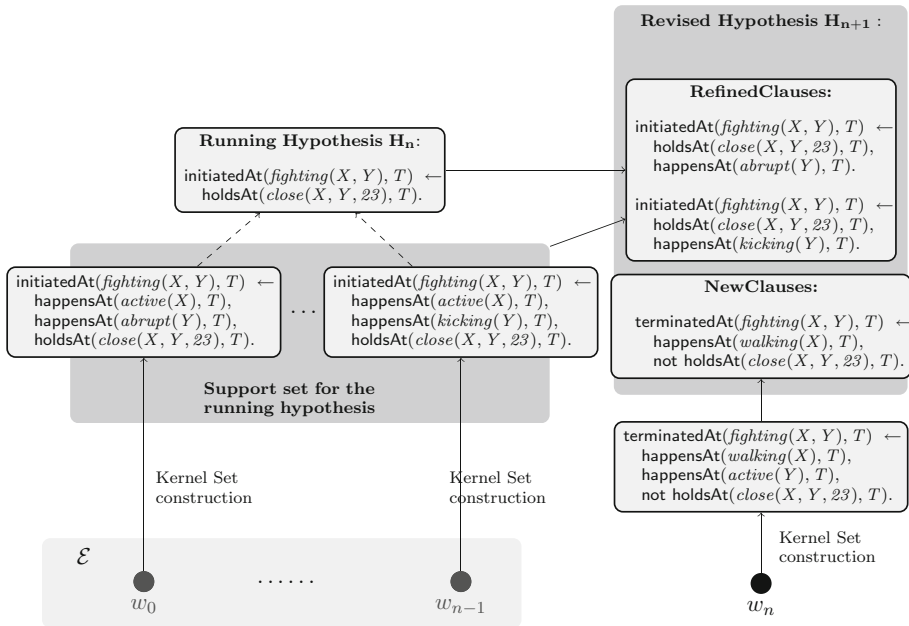
Fig. 1 Revision of a hypothesis H_n in response to a new example window w_n. $\mathcal{E}$ represents the historical memory of examples

A proof of soundness and the single-pass revision strategy of ILED is given in Proposition 3, "Appendix 2". The Pseudocode of ILED's strategy is provided in Algorithm 3, "Appendix 2".

3.1 Support set

In order to define the support set, we use the notion of most-specific clause. Given a set of mode declarations M, a clause C in the mode language $\mathcal{L}(M)$ (see "Appendix 1" for a formal definition) is most-specific if it does not θ-subsume any other clause in $\mathcal{L}(M)$. θ-subsumption is defined below.

Definition 2 (θ-*subsumption*) Clause C θ-subsumes clause D, denoted $C \preceq D$, if there exists a substitution θ such that $head(C)\theta = head(D)$ and $body(C)\theta \subseteq body(D)$, where $head(C)$ and $body(C)$ denote the head and the body of clause C respectively. Program Π_1 θ-subsumes program Π_2 if for each clause $C \in \Pi_1$ there exists a clause $D \in \Pi_2$ such that $C \preceq D$.

Intuitively, the support set of a clause C is a "bottom program" that consists of most-specific versions of the clauses that disjunctively define the concept captured by C. A formal account is given in Definition 3.

Definition 3 (*Support set*) Let $\mathcal{E}$ be the historical memory, M a set of mode declarations, $\mathcal{L}(M)$ the corresponding mode language of M and $C \in \mathcal{L}(M)$ a clause. Also, let us denote by $cov_{\mathcal{E}}(C)$ the coverage of clause C in the historical memory, i.e. $cov_{\mathcal{E}}(C) = \{e \in \mathcal{E} \mid \mathsf{SDEC} \cup C \vDash e\}$. The support set $C.supp$ of clause C is defined as follows:

Algorithm 1 Support set construction and maintenance

1: **let** $w_n \notin \mathcal{E}$ be an example window, H_n a current hypothesis and
   $H'_n = NewClauses \cup RefinedClauses \cup RetainedClauses$ a revision of H_n, generated in w_n.
2: **for all** $C \in H'_n$ **do**
3:     **if** $C \in NewClauses$ **then**
4:         $C.supp \leftarrow \{D \in K \mid C \preceq D\}$, where K is the **variablized Kernel Set** of w_n
           from which *NewClauses* is generated.
5:     **else if** $C \in RefinedClauses$ **then**
6:         $C.supp \leftarrow \{D \in C_{parent}.supp \mid C \preceq D\}$, where C_{parent} is the "ancestor"
           clause of C, i.e. the clause from which C results by specialization.
7:     **else**
8:         **let** $e_C^{w_n}$ be the *true positives* that C covers in w_n, if C is an initiatedAt clause, or
             the *true negatives* that C covers, if it is a terminatedAt clause.
9:         **if** SDEC $\cup$ $C.supp \nvDash e_C^{w_n}$ **then**
10:            **let** K be a **variablized Kernel Set** of w_n.
11:            $C.supp \leftarrow C.supp \cup K'$, where $K' \subseteq K$, such that SDEC $\cup K' \vDash e_C^{w_n}$

$$C.supp = \bigcup_{e \in cov_\mathcal{E}(C)} \{D \in \mathcal{L}(M) \mid e \in cov_\mathcal{E}(D) \text{ and } C \preceq D \text{ and}$$
$$\forall D' \in \mathcal{L}(M), \text{ if } e \in cov_\mathcal{E}(D') \text{ then } D' \preceq D\}$$

The support set of clause C is thus defined as the set consisting of one *bottom clause* (Muggleton 1995) per each example $e \in cov_\mathcal{E}(C)$, i.e. one most-specific clause D of $\mathcal{L}(M)$ such that $C \preceq D$ and SDEC $\cup D \vDash e$. Assuming no length bounds on hypothesized clauses, each such bottom clause is unique[2] and covers at least one example from $cov_E(C)$; note that since the bottom clauses for a set of examples in $cov_\mathcal{E}(C)$ may coincide (i.e. be θ-subsumption equivalent – they θ-subsume each other), a clause D in $C.supp$ may cover more than one example from $cov_\mathcal{E}(C)$. Proposition 1 highlights the main property of the structure. The proof is given in "Appendix 2".

Proposition 1 *Let C be a clause in $\mathcal{L}(M)$. $C.supp$ is the most specific program of $\mathcal{L}(M)$ such that $cov_\mathcal{E}(C.supp) = cov_\mathcal{E}(C)$.*

Proposition 1 implies that clause C and its support set $C.supp$ define a space $\mathcal{S}$ of specializations of C, each of which is bound by a most-specific specialization, among those that cover the positive examples that C covers. In other words, for every $D \in \mathcal{S}$ there is a $C_s \in C.supp$ so that $C \preceq D \preceq C_s$ and C_s covers at least one example from $cov_\mathcal{E}(C)$. Moreover, Proposition 1 ensures that space $\mathcal{S}$ contains refinements of clause C that collectively preserve the coverage of C in the historical memory. The purpose of $C.supp$ is thus to serve as a search space for refinements R_C of clause C for which $C \preceq R_C \preceq C.supp$ holds. Since such refinements preserve C's coverage of positive examples, clause C may be refined w.r.t. a window w_n, avoiding the overhead of re-testing the refined program on $\mathcal{E}$ for completeness. However, to ensure that the support set can indeed be used as a refinement search space, one must ensure that $C.supp$ will always contain such a refinement R_C. This proof is provided in Proposition 2, "Appendix 9".

The construction of the support set, presented in Algorithm 1, is a process that starts when C is added in the running hypothesis and continues as long as new example windows

---

[2] The bottom clause relative to an example can be large, or even infinite. To constrain its size, several restrictions are imposed on the language, such as a maximum clause length, or a maximum variable depth. We refrain from assuming extra language bias related to clause length and instead, for the purposes of this work, we assume a finite domain and impose no particular bounds on clause length. In such context, the bottom clause of an example e is unique and results from the *ground* most-specific clause that covers e, by properly replacing terms with variables, as indicated by the mode declarations.

Table 4 Knowledge for Example 2

Window w_1	
Narrative	**Annotation**
happensAt($active(id_1)$, 10).	not holdsAt($fighting(id_1, id_2)$, 10).
happensAt($abrupt(id_2)$, 10).	holdsAt($fighting(id_1, id_2)$, 11).
holdsAt($close(id_1, id_2, 23)$, 10).	
Kernel Set	**Variabilized Kernel Set**
initiatedAt($fighting(id_1, id_2)$, 10) ←	$K_1 =$ initiatedAt($fighting(X, Y), T$) ←
happensAt($active(id_1)$, 10),	happensAt($active(X), T$),
happensAt($abrupt(id_2)$, 10)	happensAt($abrupt(Y), T$),
holdsAt($close(id_1, id_2, 23)$, 10)	holdsAt($close(X, Y, 23), T$).
Running Hypothesis	**Support Set**
$C =$ initiatedAt($fighting(X, Y), T$) ←	$C.supp = \{K_1\}$
happensAt($active(X), T$).	
Window w_2	
Narrative	**Annotation**
happensAt($active(id_1)$, 20).	not holdsAt($fighting(id_1, id_2)$, 20).
happensAt($kicking(id_2)$, 20).	holdsAt($fighting(id_1, id_2)$, 21).
holdsAt($close(id_1, id_2, 23)$, 20).	
Kernel Set	**Variabilized Kernel Set**
initiatedAt($fighting(id_1, id_2)$, 20) ←	$K_2 =$ initiatedAt($fighting(X, Y), T$) ←
happensAt($active(id_1)$, 20),	happensAt($active(X), T$),
happensAt($kicking(id_2)$, 20)	happensAt($kicking(Y), T$),
holdsAt($close(id_1, id_2, 23)$, 20)	holdsAt($close(X, Y, 23), T$).
Running Hypothesis	**Support Set**
Remains unchanged	$C.supp = \{K_1, K_2\}$
Window w_3	
Narrative	**Annotation**
happensAt($active(id_1)$, 30).	not holdsAt($fighting(id_1, id_2)$, 30).
happensAt($walking(id_2)$, 30).	not holdsAt($fighting(id_1, id_2)$, 31).
not holdsAt($close(id_1, id_2, 23)$, 30).	
Revised Hypothesis	**Support Set**
$C_1 =$ initiatedAt($fighting(X, Y), T$) ←	$C_1.supp = \{K_1, K_2\}$
happensAt($active(X), T$),	
holdsAt($close(X, Y, 23), T$).	

arrive. While this happens, clause C may be refined or retained, and its support set is updated accordingly. The details of Algorithm 1 are presented in Example 2, which also demonstrates how ILED processes incoming examples and revises hypotheses.

Example 2 Consider the annotated examples and running hypothesis related to the *fighting* high-level event from the activity recognition application shown in Table 4. We assume that ILED starts with an empty hypothesis and an empty historical memory, and that w_1 is the first input example window. The currently empty hypothesis does not cover the provided

examples, since in w_1 *fighting* between persons id_1 and id_2 is initiated at time 10 and thus holds at time 11. Hence ILED starts the process of generating an initial hypothesis. In the case of an empty hypothesis, ILED reduces to XHAIL and operates on a Kernel Set of w_1 only. The variabilized Kernel Set in this case will be the single-clause program K_1 presented in Table 4, generated from the corresponding ground clause. Generalizing this Kernel Set yields a minimal hypothesis that covers w_1. One such hypothesis is clause C shown in Table 4. ILED stores w_1 in $\mathcal{E}$ and initializes the support set of the newly generated clause C as in line 3 of Algorithm 1, by selecting from K_1 the clauses that are θ-subsumed by C, in this case, K_1's single clause.

Window w_2 arrives next. In w_2, *fighting* is initiated at time 20 and thus holds at time 21. The running hypothesis correctly accounts for that and thus no revision is required. However, $C.supp$ does not cover w_2 and unless proper actions are taken, property (i) of Proposition 1 will not hold once w_2 is stored in $\mathcal{E}$. ILED thus generates a new Kernel Set K_2 from window w_2, as presented in Table 4, and updates $C.supp$ as shown in lines 7–11 of Algorithm 1. Since C θ-subsumes K_2, the latter is added to $C.supp$, which now becomes $C.supp = \{K_1, K_2\}$. Now $cov_{\mathcal{E}}(C.supp) = cov_{\mathcal{E}}(C)$, hence in effect, $C.supp$ is a summarization of the coverage of clause C in the historical memory.

Window w_3 arrives next, which has no positive examples for the initiation of *fighting*. The running hypothesis is revisable in window w_3, since clause C covers a negative example at time 31, by means of initiating the fluent *fighting(id_1, id_2)* at time 30. To address the issue, ILED searches $C.supp$, which now serves as a refinement search space, to find a refinement R_C that rejects the negative example, and moreover $R_C \preceq C.supp$. Several choices exist for that. For instance, the following program

initiatedAt($fighting(X, Y), T$) ←
     happensAt($active(X), T$),
     happensAt($abrupt(Y), T$).
	initiatedAt($fighting(X, Y), T$) ←
	     happensAt($active(X), T$),
	     happensAt($kicking(Y), T$).

is such a refinement R_C, since it does not cover the negative example in w_3 and subsumes $C.supp$. ILED however is biased towards minimal theories, in terms of the overall number of literals and would prefer the more compressed refinement C_1, shown in Table 4, which also rejects the negative example in w_3 and subsumes $C.supp$. Clause C_1 replaces the initial clause C in the running hypothesis. The hypothesis now becomes complete and consistent w.r.t. $\mathcal{E}$. Note that the hypothesis was refined by local reasoning only, i.e. reasoning within window w_3 and the support set, avoiding costly look-back in the historical memory. The support set of the new clause C_1 is initialized (line 5 of Algorithm 1), by selecting the subset of the support set of its parent clause that is θ-subsumed by C_1. In this case $C_1 \preceq C.supp = \{K_1, K_2\}$, hence $C_1.supp = C.supp$. □

The support set of a clause C is a compressed enumeration of the examples that C covers throughout the historical memory. It is compressed because each variabilized clause in the set is expected to encode many examples. In contrast, a ground version of the support set would be a plain enumeration of examples, since in the general case, it would require one ground clause per example. The main advantage of the "lifted" character of the support set over a plain enumeration of the examples is that it requires much less memory to encode the necessary information, an important feature in large-scale (temporal) applications. Moreover, given that training examples are typically characterized by heavy repetition, abstracting away

Algorithm 2 revise(SDEC, H_n, w_n, $K_v^{w_n}$)

Input: *The axioms of* SDEC, *a running hypothesis H_n an example window w_n and a variabilized Kernel Set $K_v^{w_n}$ of w_n.*

Output: *A revised hypothesis H_n'*

1: **let** $U(K_v^{w_n}, H_n) \leftarrow$ GeneralizationTransformation$(K_v^{w_n}) \cup$ RefinementTransformation(H_n)
2: **let** Φ be the abductive task $\Phi = ALP(\text{SDEC} \cup U(K_v^{w_n}, H_n), \{use/2, use/3\}, w_n)$
3: **if** Φ has a solution **then**
4:     **let** Δ be a minimal solution of Φ
5:     **let** $NewClauses = \{\alpha_i \leftarrow \delta_i^1 \wedge \ldots \wedge \delta_i^n \mid$
               α_i is the head of the i−th clause $C_i \in K_v^{w_n}$
               and δ_i^j is the j−th body literal of C_i
               and $use(i, 0) \in$ and $use(i, j) \in$, $1 \leq j \leq n$ }
6:     **let** $RefinedClauses = \{ head(C_i) \leftarrow body(C_i) \wedge \delta_i^{j,k_1} \wedge \ldots \wedge \delta_i^{j,k_m} \mid$
               $C_i \in H_n$ and $use(i, j, k_l) \in$, $1 \leq l \leq m, 1 \leq j \leq |C_i.supp|$ }
7:     **let** $RetainedClauses = \{C_i \in H_n \mid use(i, j, k) \notin$ for any $j, k\}$
8:     **let** $RefinedClauses =$ ReduceRefined$(NewClauses, RefinedClauses, RetainedClauses)$
9: **else**
10:    **Return** No Solution
11: **Return** $\langle RetainedClauses, RefinedClauses, NewClauses \rangle$

Table 5 Syntactic transformations performed by ILED

GeneralizationTransformation	RefinementTransformation
Input: A variabilized Kernel set K_v	**Input:** A running hypothesis H_n
For each clause $D_i = \alpha_i \leftarrow \delta_i^1, \ldots, \delta_i^n \in F_v$:	**For each** clause $D_i \in H_n$:
Add an extra atom $use(i, 0)$ to the body of D_i	**For each** clause $\Gamma_i^j \in D_i.supp$
and replace each body literal δ_i^j with a new	Generate one clause
atom of the form $try(i, j, v(\delta_i^j))$, where $v(\delta_i^j)$	$\alpha_i \leftarrow body(D_i) \wedge$ not $exception(i, j, v(\alpha_i))$
contains the variables that appear in δ_i^j.	where α_i is the head of D_i and $v(\alpha_i)$
Generate two new clauses of the form	contains its variables. Generate one clause
$try(i, j, v(\delta_i^j)) \leftarrow use(i, j), \delta_i^j$ and	$exception(i, j, v(a_i)) \leftarrow use(i, j, k),$ not $\delta_i^{j,k}$
$try(i, j, v(\delta_i^j)) \leftarrow$ not $use(i, j)$ for each δ_i^j.	for each body literal $\delta_i^{j,k}$ of Γ_i^j.

redundant parts of the search space results in a memory structure that is expected to grow in size slowly, allowing for fast search that scales to a large amount of historical data.

3.2 Implementing revisions

Algorithm 2 presents the revision function of ILED. The input consists of SDEC as background knowledge, a running hypothesis H_n, an example window w_n and a variabilized Kernel Set $K_v^{w_n}$ of w_n. The clauses of $K_v^{w_n}$ and H_n are subject to the GeneralizationTansformation and the RefinementTransformation respectively, presented in Table 5. The former is the transformation discussed in Sect. 2.2.1, that turns the Kernel Set into a defeasible program, allowing the construction of new clauses. The RefinementTransformation aims at the refinement of the clauses of H_n using their support sets. It involves two fresh predicates, *exception/3* and *use/3*. For each clause $D_i \in H_n$ and for each of its support set clauses $\Gamma_i^j \in D_i.supp$, one new clause $head(D_i) \leftarrow body(D_i) \wedge$ not $exception(i, j, v(head(D_i)))$ is generated, where $v(head(D_i))$ is a term that contains the variables of $head(C_i)$. Then an

additional clause $exception(i, j, v(head(D_i))) \leftarrow use(i, j, k) \wedge$ not $\delta_i^{j,k}$ is generated, for each body literal $\delta_i^{j,k} \in \Gamma_i^j$.

The syntactically transformed clauses are put together in a program $U(K_v^{w_n}, H_n)$ (line 1 of Algorithm 2), which is used as a background theory along with SDEC. A minimal set of $use/2$ and $use/3$ atoms is abduced as a solution to the abductive task Φ in line 2 of Algorithm 2. Abduced $use/2$ atoms are used to construct a set of *NewClauses*, as discussed in Sect. 2.2.1 (line 5 of Algorithm 2). These new clauses account for some of the examples in w_n, which cannot be covered by existing clauses in H_n. The abduced $use/3$ atoms indicate clauses of H_n that must be refined. From these atoms, a refinement R_{D_i} is generated for each incorrect clause $D_i \in H_n$, such that $D_i \preceq R_{D_i} \preceq D_i.supp$ (line 6 of Algorithm 2). Clauses that lack a corresponding $use/3$ atom in the abductive solution are retained (line 7 of Algorithm 2).

The intuition behind refinement generation is as follows: Assume that clause $D_i \in H_n$ must be refined. This can be achieved by means of the extra clauses generated by the RefinementTransformation. These clauses provide definitions for the exception atom, namely one for each body literal in each clause of $D_i.supp$. From these clauses, one can satisfy the exception atom by satisfying the complement of the corresponding support set literal and abducing the accompanying $use/3$ atom. Since an abductive solution Δ is minimal, the abduced $use/3$ atoms correspond precisely to the clauses that must be refined.

Hence, each inconsistent clause $D_i \in H_n$ and each $\Gamma_{i^j \in D_i.supp}$ correspond to a set of abduced $use/3$ atoms of the form $use(i, j, k_1), \ldots, use(i, j, k_n)$. These atoms indicate that a specialization of D_i may be generated by adding to the body of D_i the literals $\delta_i^{j,k_1}, \ldots, \delta_i^{j,k_n}$ from Γ_i^j. Then a refinement R_{D_i} such that $D_i \preceq R_{D_i} \preceq D_i.supp$ may be generated by selecting one specialization of clause D_i from each support set clause in $D_i.supp$.

Example 3 Table 6 presents the process of ILED's refinement. The annotation lacks positive examples and the running hypothesis consists of a single clause C, with a support set of two clauses. Clause C is inconsistent since it entails two negative examples, namely holdsAt(*fighting*(id_1, id_2), *2*) and holdsAt(*fighting*(id_3, id_4), *3*). The program that results by applying the RefinementTransformation to the support set of clause C is presented in Table 6, along with a minimal abductive explanation of the examples, in terms of $use/3$ atoms. Atoms $use(1, 1, 2)$ and $use(1, 1, 3)$ correspond respectively to the second and third body literals of the first support set clause, which are added to the body of clause C, resulting in the first specialization presented in Table 6. The third abduced atom $use(1, 2, 2)$ corresponds to the second body literal of the second support set clause, which results in the second specialization in Table 6. Together, these specializations form a refinement of clause C that subsumes $C.supp$.   □

Minimal abductive solutions imply that the running hypothesis is minimally revised. Revisions are minimal w.r.t. the length of the clauses in the revised hypothesis, but are not minimal w.r.t. the number of clauses, since the refinement strategy described above may result in refinements that include redundant clauses: Selecting one specialization from each support set clause to generate a refinement of a clause is sub-optimal, since there may exist other refinements with fewer clauses that also subsume the whole support set, as Example 2 demonstrates. To avoid unnecessary increase of the hypothesis size, the generation of refinements is followed by a "reduction" step (line 8 of Algorithm 2). The ReduceRefined function works as follows. For each refined clause C, it first generates all possible refinements from $C.supp$. This can be realized with the abductive refinement technique described above. The only difference is that the abductive solver is instructed to find all abductive explanations in terms of $use/3$ atoms, instead of one. Once all refinements are generated, ReduceRefined

Table 6 Clause refinement by ILED

Input	
Narrative	**Annotation**
happensAt($abrupt(id_1)$, 1).	not holdsAt($fighting(id_1, id_2)$, 1).
happensAt($inactive(id_2)$, 1).	not holdsAt($fighting(id_3, id_4)$, 1).
holdsAt($close(id_1, id_2, 23)$, 1).	not holdsAt($fighting(id_1, id_2)$, 2).
happensAt($abrupt(id_3)$, 2).	not holdsAt($fighting(id_3, id_4)$, 2).
happensAt($abrupt(id_4)$, 2).	not holdsAt($fighting(id_1, id_2)$, 3).
not holdsAt($close(id_3, id_4, 23)$, 2).	not holdsAt($fighting(id_3, id_4)$, 3).
Running hypothesis	**Support set**
C = initiatedAt($fighting(X, Y)$, T) ←	C_s^1 = initiatedAt($fighting(X, Y)$, T) ←
happensAt($abrupt(X)$, T).	happensAt($abrupt(X)$, T),
	happensAt($abrupt(Y)$, T),
	holdsAt($close(X, Y, 23)$, T).
	C_s^2 = initiatedAt($fighting(X, Y)$, T) ←
	happensAt($abrupt(X)$, T),
	happensAt($active(Y)$, T),
	holdsAt($close(X, Y, 23)$, T).
Refinement transformation:	
From C_s^1 :	**From C_s^2 :**
initiatedAt($fighting(X, Y)$, T) ←	initiatedAt($fighting(X, Y)$, T) ←
happensAt($abrupt(X)$, T),	happensAt($abrupt(X)$, T),
not $exception(1, 1, vars(X, Y, T))$.	not $exception(1, 2, vars(X, Y, T))$.
$exception(1, 1, vars(X, Y, T))$ ←	$exception(1, 2, vars(X, Y, T))$ ←
$use(1, 1, 2)$, not happensAt($abrupt(Y)$, T).	$use(1, 2, 2)$, not happensAt($active(Y)$, T).
$exception(1, 1, vars(X, Y, T))$ ←	$exception(1, 2, vars(X, Y, T))$ ←
$use(1, 1, 3)$, not holdsAt($close(X, Y, 23)$, T).	$use(1, 2, 3)$, not holdsAt($close(X, Y, 23)$, T).
Minimal abductive solution	**Generated refinements**
$\Delta = \{use(1, 1, 2), use(1, 1, 3), use(1, 2, 2)\}$	initiatedAt($fighting(X, Y)$, T) ←
	happensAt($abrupt(X)$, T),
	happensAt($abrupt(Y)$, T),
	holdsAt($close(X, Y, 23)$, T).
	initiatedAt($fighting(X, Y)$, T) ←
	happensAt($abrupt(X)$, T),
	happensAt($active(Y)$, T).

searches the revised hypothesis, augmented with all refinements of clause C, to find a reduced set of refinements of C that subsume $C.supp$.

4 Discussion

Like XHAIL, ILED aims at soundness, that is, hypotheses which cover all given examples. XHAIL ensures soundness by generalizing all examples in one go. In contrast, ILED has access to a memory of past experience for which newly acquired knowledge must

account. Concerning completeness, XHAIL is a state-of-the-art system among its Inverse Entailment-based peers. Although ILED preserves XHAIL's soundness, it does not preserve its completeness properties, due to the fact that ILED operates incrementally to gain efficiency. Thus there are cases where a hypothesis can be discovered by XHAIL, but be missed by ILED. As an example, consider cases where a target hypothesis captures long-term temporal relations in the data, as for instance, in the following clause:

$$\text{initiatedAt}(moving(X, Y), T) \leftarrow$$
$$\text{happensAt}(walking(Y), T1),$$
$$T1 < T.$$

In such cases, if the parts of the data that are connected via a long-range temporal relation are given in different windows, ILED has no way to correlate these parts in order to discover the temporal relation. However, one can always achieve XHAIL's functionality by increasing appropriately ILED's window size.

An additional trade-off for efficiency is that not all of ILED's revisions are fully evaluated on the historical memory. For instance, selecting a particular clause in order to cover a new example, may result in a large number of refinements and an unnecessarily lengthy hypothesis, as compared to one that may have been obtained by selecting a different initial clause. On the other hand, fully evaluating all possible choices over $\mathcal{E}$ requires extensive inference. Thus simplicity and compression of hypotheses in ILED have been sacrificed for efficiency.

In ILED, a large part of the theorem proving effort that is involved in clause refinement reduces to computing subsumption between clauses, which is a hard task. Moreover, just as the historical memory grows over time, so do (in the general case) the support sets of the clauses in the running hypothesis, increasing the cost of computing subsumption. However, as in principle the largest part of a search space is redundant and the support set focuses only on its interesting parts, one would not expect that the support set will grow to a size that makes subsumption computation less efficient than inference over the entire $\mathcal{E}$. In addition, a number of optimization techniques have been developed over the years and several generic subsumption engines have been proposed (Maloberti and Sebag 2004; Kuzelka and Zelezny 2008; Santos and Muggleton 2010), some of which are able to efficiently compute subsumption relations between clauses comprising thousands of literals and hundreds of distinct variables.

5 Experimental evaluation

In this section, we present experimental results from two real-world applications: Activity recognition, using real data from the benchmark CAVIAR video surveillance dataset,[3] as well as large volumes of synthetic CAVIAR data; and City Transport Management (CTM) using data from the PRONTO[4] project.

Part of our experimental evaluation aims to compare ILED with XHAIL. To achieve this aim we had to implement XHAIL, because the original implementation was not publicly available until recently (Bragaglia and Ray 2014). All experiments were conducted on a 3.2 GHz Linux machine with 4 GB of RAM. The algorithms were implemented in Python,

---

[3] http://homepages.inf.ed.ac.uk/rbf/CAVIARDATA1/.

[4] http://www.ict-pronto.org/.

using the Clingo[5] Answer Set Solver (Gebser et al. 2012) as the main reasoning component, and a Mongodb[6] NoSQL database for the historical memory of the examples. The code and datasets used in these experiments can be downloaded from https://github.com/nkatzz/ILED.

5.1 Activity recognition

In activity recognition, our goal is to learn definitions of high-level events, such as *fighting, moving* and *meeting*, from streams of low-level events like *walking, standing, active* and *abrupt*, as well as spatio-temporal knowledge. We use the benchmark CAVIAR dataset for experimentation. Details on the CAVIAR dataset can be found in Artikis et al. (2010).

CAVIAR contains noisy data mainly due to human errors in the annotation (List et al. 2005; Artikis et al. 2010). Thus, for the experiments we manually selected a noise-free subset of CAVIAR. The resulting dataset consists of 1000 examples (that is, data for 1000 distinct time points) concerning the high-level events *moving*, *meeting* and *fighting*. These data, selected from different parts of the CAVIAR dataset, were combined into a continuous annotated stream of narrative atoms, with time ranging from 0 to 1000.

In addition to the real data, we generated synthetic data based on the manually-developed CAVIAR event definitions described in Artikis et al. (2010). In particular, streams of low-level events were created randomly and were then classified using the rules of Artikis et al. (2010). The generated data consists of approximately 10^5 examples, which amounts to 100 MB of data.

The synthetic data is much more complex than the real CAVIAR data. This is due to two main reasons: First, the synthetic data includes significantly more initiations and terminations of a high-level event, thus much larger learning effort is required to explain it. Second, in the synthetic dataset more than one high-level event may be initiated or terminated at the same time point. This results in Kernel Sets with more clauses, which are hard to generalize simultaneously.

5.1.1 ILED versus XHAIL

The purpose of this experiment was to assess whether ILED can efficiently generate hypotheses comparable in size and predictive quality to those of XHAIL. To this end, we compared both systems on real and synthetic data using tenfold cross validation with replacement. For the real data, 90 % of randomly selected examples, from the total of 1000 were used for training, while the remaining 10 % was retained for testing. At each run, the training data were presented to ILED in example windows of sizes 10, 50, 100. The data were presented in one batch to XHAIL. For the synthetic data, 1000 examples were randomly sampled at each run from the dataset for training, while the remaining data were retained for testing. Similar to the real data experiments, ILED operated on windows of sizes of 10, 50, 100 examples and XHAIL on a single batch.

Table 7 presents the experimental results. Training times are significantly higher for XHAIL, due to the increased complexity of generalizing Kernel Sets that account for the whole set of the presented examples at once. These Kernel Sets consisted, on average, of 30–35 16-literal clauses, in the case of the real data, and 60–70 16-literal clauses in the case of the synthetic data. In contrast, ILED had to deal with much smaller Kernel Sets. The complexity of abductive search affects ILED as well, as the size of the input windows grows.

---

[5] http://potassco.sourceforge.net/.

[6] http://www.mongodb.org/.

Table 7 Comparison of ILED and XHAIL

	ILED			XHAIL
	$G = 10$	$G = 50$	$G = 100$	$G = 900$
Real CAVIAR data				
Training time (s)	34.15 (±6.87)	23.04 (±13.50)	286.74 (±98.87)	1560.88 (±4.24)
Revisions	11.2 (±3.05)	9.1 (±0.32)	5.2 (±2.1)	–
Hypothesis size	17.82 (±2.18)	17.54 (±1.5)	17.5 (±1.43)	15 (±0.067)
Precision	98.713 (±0.052)	99.767 (±0.038)	99.971 (±0.041)	99.973 (±0.028)
Recall	99.789 (±0.083)	99.845 (±0.32)	99.988 (±0.021)	99.992 (±0.305)
Synthetic CAVIAR data				
Training time (s)	38.92 (±9.15)	33.87 (±9.74)	468 (±102.62)	21429 (±342.87)
Revisions	28.7 (±9.34)	15.4 (±7.5)	12.2 (±6.23)	–
Hypothesis size	143.52 (±19.14)	138.46 (±22.7)	126.43 (±15.8)	118.18 (±14.48)
Precision	55.713 (±0.781)	57.613 (±0.883)	63.236 (±0.536)	63.822 (±0.733)
Recall	68.213 (±0.873)	71.813 (±0.756)	71.997 (±0.518)	71.918 (±0.918)

G is the window granularity

ILED handles the learning task relatively well (in approximately 30 s) when the examples are presented in windows of 50 examples, but the training time increases almost 15 times if the window size is doubled.

Concerning the size of the produced hypothesis, the results show that in the case of real CAVIAR data, the hypotheses constructed by ILED are comparable in size with a hypothesis constructed by XHAIL. In the case of synthetic data, the hypotheses returned by both XHAIL and ILED were significantly more complex. Note that for ILED the hypothesis size decreases as the window size increases. This is reflected in the number of revisions that ILED performs, which is significantly smaller when the input comes in larger batches of examples. In principle, the richer the input, the better the hypothesis that is initially acquired, and consequently, the less the need for revisions in response to new training instances. There is a trade-off between the window size (thus the complexity of the abductive search) and the number of revisions. A small number of revisions on complex data (i.e. larger windows) may have a greater total cost in terms of training time, as compared to a greater number of revisions on simpler data (i.e. smaller windows). For example, in the case of window size 100 for the real CAVIAR data, ILED performs 5 revisions on average and requires significantly more time than in the case of a window size 50, where it performs 9 revisions on average. On the other hand, training times for windows of size 50 are slightly better than those obtained when the examples are presented in smaller windows of size 10. In this case, the "unit cost" of performing revisions w.r.t a single window are comparable between windows of size 10 and 50. Thus the overall cost in terms of training time is determined by the total number of revisions, which is greater in the case of window size 10.

Concerning predictive quality, the results indicate that ILED's precision and recall scores are comparable to those of XHAIL. For larger input windows, precision and recall are almost the same as those of XHAIL. This is because ILED produces better hypotheses from larger input windows. Precision and recall are smaller in the case of synthetic data for both systems, because the testing set in this case is much larger and complex than in the case of real data.
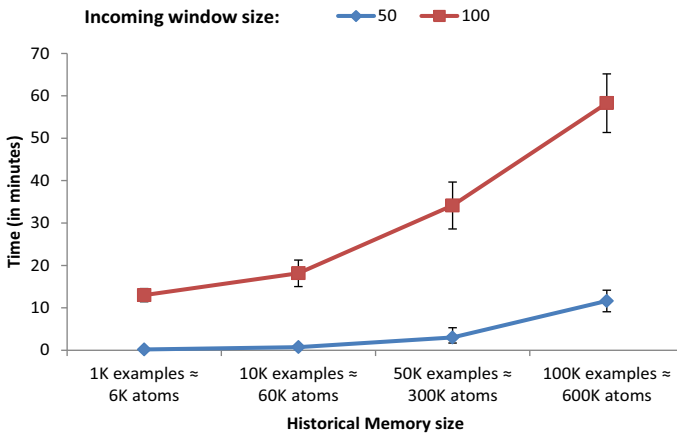
Fig. 2 Average times needed for ILED to revise an initial hypothesis in the face of new evidence presented in windows of size 10, 50 and 100 examples. The initial hypothesis was obtained from a training set of varying size (1K, 10K, 50K and 100K examples) which subsequently served as the historical memory

5.1.2 ILED scalability

The purpose of this experiment was to assess the scalability of ILED. The experimental setting was as follows: Sets of examples of varying sizes were randomly sampled from the synthetic dataset. Each such example set was used as a training set in order to acquire an initial hypothesis using ILED. Then a new window which did not satisfy the hypothesis at hand was randomly selected and presented to ILED, which subsequently revised the initial hypothesis in order to account for both the historical memory (the initial training set) and the new evidence. For historical memories ranging from 10^3 to 10^5 examples, a new training window of size 10, 50 and 100 was selected from the whole dataset. The process was repeated ten times for each different combination of historical memory and new window size. Figure 2 presents the average revision times. The revision times for new window sizes of 10 and 50 examples are very close and therefore omitted to avoid clutter. The results indicate that revision time grows polynomially in the size of the historical memory.

5.2 City transport management

In this section we present experimental results from the domain of City Transport Management (CTM), using data from the PRONTO[7] project. In PRONTO, the goal was to inform the decision-making of transport officials by recognising high-level events related to the punctuality of a public transport vehicle (bus or tram), passenger/driver comfort and safety. These high-level events were requested by the public transport control centre of Helsinki, Finland, in order to support resource management. Low-level events were provided by sensors installed in buses and trams, reporting on changes in position, acceleration/deceleration, in-vehicle temperature, noise level and passenger density. At the time of the project, the available datasets included only a subset of the anticipated low-level event types as some low-level event detection components were not functional. Therefore, a synthetic dataset was generated. The synthetic PRONTO data has proven to be considerably more challenging for event recognition than the real data (Artikis et al. 2015), and therefore we chose the former for
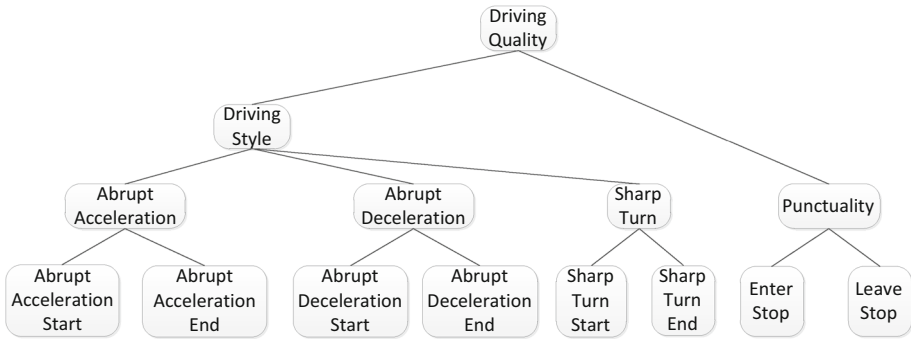
---

[7] http://www.ict-pronto.org/.

Fig. 3 City Transport Management partial event hierarchy (we omit the whole hierarchy to save space). Additional high-level events, not presented here are *noise level*, *vehicle temperature*, and *passenger density*, which depend on corresponding low-level events and affect *driving quality*

evaluating ILED. The CTM dataset contains $5 \cdot 10^4$ examples, which amount approximately to 70 MB of data.

In contrast to the activity recognition application, the manually developed event definitions of CTM form a hierarchy. In these definitions, it is possible to define a function level that maps high-level events to non-negative integers as follows: A level-1 event is defined in terms of low-level events (input data) only. A level-n event is defined in terms of at least one level-$n − 1$ event and a possibly empty set of low-level events and high-level events of level below $n − 1$. Hierarchical definitions are significantly more complex to learn compared to non-hierarchical ones. This is because initiations and terminations of events in the lower levels of the hierarchy appear in the bodies of event definitions in the higher levels, hence all target definitions must be learnt simultaneously. As we show in the experiments, this has a striking effect on the learning effort. A solution for simplifying the learning task is to utilize knowledge about the domain (the hierarchy), learn event definitions separately, and use the acquired theories from lower levels of the hierarchy as non-revisable background knowledge when learning event definitions for the higher levels. A part of The CTM hierarchy is presented in Fig. 3. Consider the following fragment:

$$\mathsf{initiatedAt}(punctuality(Id, nonPunctual), T) \leftarrow$$
$$\quad \mathsf{happensAt}(stopEnter(Id, StopId, late), T). \tag{1}$$
$$\mathsf{initiatedAt}(punctuality(Id, nonPunctual), T) \leftarrow$$
$$\quad \mathsf{happensAt}(stopLeave(Id, StopId, early), T). \tag{2}$$
$$\mathsf{terminatedAt}(punctuality(Id, nonPunctual), T) \leftarrow$$
$$\quad \mathsf{happensAt}(stopEnter(Id, StopId, early), T). \tag{3}$$
$$\mathsf{terminatedAt}(punctuality(Id, nonPunctual), T) \leftarrow$$
$$\quad \mathsf{happensAt}(stopEnter(Id, StopId, scheduled), T). \tag{4}$$
$$\mathsf{initiatedAt}(drivingQuality(Id, low), T) \leftarrow$$
$$\quad \mathsf{initiatedAt}(punctuality(Id, nonPunctual), T),$$
$$\quad \mathsf{holdsAt}(drivingStyle(Id, unsafe), T). \tag{5}$$
$$\mathsf{initiatedAt}(drivingQuality(Id, low), T) \leftarrow$$
$$\quad \mathsf{initiatedAt}(drivingStyle(Id, unsafe), T),$$

$$holdsAt(punctuality(Id, nonPunctual), T). \tag{6}$$

$$terminatedAt(drivingQuality(Id, low), T) \leftarrow$$
$$terminatedAt(punctuality(Id, nonPunctual), T). \tag{7}$$

$$terminatedAt(drivingQuality(Id, low), T) \leftarrow$$
$$terminatedAt(drivingStyle(Id, unsafe), T). \tag{8}$$

Clauses (1) and (2) state that a period of time for which vehicle *Id* is said to be *non-punctual* is initiated if it enters a stop later, or leaves a stop earlier than the scheduled time. Clauses (3) and (4) state that the period for which vehicle *Id* is said to be non-punctual is terminated when the vehicle arrives at a stop earlier than, or at the scheduled time. The definition of non-punctual vehicle uses two low-level events, *stopEnter* and *stopLeave*.

Clauses (5)–(8) define *low driving quality*. Essentially, driving quality is said to be low when the driving style is unsafe and the vehicle is non-punctual. Driving quality is defined in terms of high-level events (we omit the definition of driving style to save space). Therefore, the bodies of the clauses defining driving quality include initiatedAt/2 and terminatedAt/2 literals.

5.2.1 ILED versus XHAIL

In this experiment, we tried to learn simultaneously definitions for all target concepts, a total of nine interrelated high-level events, seven of which are level-1, one is level-2 and one is level-3. The total number of low-level events is eleven, while for both high-level and low-level events, their negations are considered during learning. According to the employed language bias, each high-level event must be learnt, while at the same time it may be present in the body of another high-level event in the form of a (potentially negated) holdsAt/2, initiatedAt/2, or terminatedAt/2 predicate.

We used tenfold cross validation with replacement, on small amounts of data, due to the complexity of the learning task. In each run of the cross validation, we randomly sampled 20 examples from the CTM dataset, 90 % of which was used for training and 10 % was retained for testing. This example size was selected after experimentation, in order for XHAIL to be able to perform in an acceptable time frame. Each sample consisted of approximately 150 atoms (narrative and annotation). The examples were given to ILED in windows of granularity 5 and 10, and to XHAIL in one batch. Table 8 presents the average training times, hypothesis size, number of revisions, precision and recall.

Table 8 Comparative performance of ILED and XHAIL on selected subsets of the CTM dataset each containing 20 examples

	ILED		XHAIL
	$G = 5$	$G = 10$	$G = 20$
Training time (h)	1.35 (±0.17)	1.88 (±0.13)	4.35 (±0.2)
Hypothesis size	28.32 (±1.19)	24.13 (±2.54)	24.02 (±0.23)
Revisions	14.78 (±2.24)	13.42 (±2.08)	–
Precision	63.344 (±5.24)	64.644 (±3.45)	66.245 (±3.83)
Recall	59.832 (±7.13)	61.423 (±5.34)	62.567 (±4.65)

G is the granularity of the windows

ILED took on average 1–2 h to complete the learning task, for windows of 5 and 10 examples, while XHAIL required more than 4 h on average to learn hypotheses from batches of 20 examples. Compared to activity recognition, the learning setting requires larger Kernel Set structures that are hard to reason with. An average Kernel Set generated from a batch of just 20 examples consisted of approximately 30–35 clauses, with 60–70 literals each.

Like the activity recognition experiments, precision and recall scores for ILED are comparable to those of XHAIL, with the latter being slightly better. Unlike the activity recognition experiments, precision and recall had a large diversity between different runs. Due to the complexity of the CTM dataset, the constructed hypotheses had a large diversity, depending on the random samples that were used for training. For example, some high-level event definitions were unnecessarily lengthy and difficult to be understood by a human expert. On the other hand, some level-1 definitions could, in some runs of the experiment, be learnt correctly even from a limited amount of data. Such definitions are fairly simple, consisting of one initiation and one termination rule, with one body literal in each case.

This experiment demonstrates several limitations of learning in large and complex applications. The complexity of the domain increases the intensity of the learning task, which in turn makes training times forbidding, even for small amount of data such as 20 examples (approximately 150 atoms). This forces one to process small sets of examples at a time, which in complex domains like CTM, results in over-fitted theories and rapid increase in hypothesis size.

5.2.2 Learning with hierarchical bias

In an effort to improve the experimental results, we utilized domain knowledge about the event hierarchy in CTM and attempted to learn high-level events in different levels separately. To do so, we had to learn a complete definition for a high-level event from the entire dataset, before utilizing it as background knowledge in the learning process of a higher-level event. To facilitate the learning task further, we also used expert knowledge about the relation between specific low-level and high-level events, excluding from the language bias mode declarations which were irrelevant to the high-level event that was being learnt at each time.

The experimental setting was therefore as follows: Starting from the level-1 target events, we processed the whole CTM dataset in windows of 10, 50 and 100 examples with ILED. Each high-level event was learnt independently of the others. Once complete definitions for all level-1 high-level events were constructed, they were added to the background knowledge. Then we proceeded with learning the definition of the single level-2 event (see Fig. 3). Finally, after successfully constructing the level-2 definition, we performed learning in the top-level of the hierarchy, using the previously constructed level-1 and level-2 event definitions as background knowledge. We did not attempt a comparison with XHAIL because it is not able to operate on the entire dataset.

Table 9 presents the results. For level-1 events, scores are presented as minimum–maximum pairs. For instance, the training times for level-1 events with windows of 10 examples, range from 4.46 to 4.88 min. Levels 2 and 3 have just one definition each, therefore Table 9 presents the respective scores from each run. Training times, hypotheses sizes and overall numbers of revisions are comparable for all levels of the event hierarchy. Level-1 event definitions were the easiest to acquire, with training times ranging approximately between 4.50 and 7 min. This was expected since clauses in level-1 definitions are significantly simpler than level-2 and level-3 ones. The level-2 event definition was the hardest to construct with training times ranging between 8 and 10 min, while a significant number of revisions was required for all window granularities. The definition of this high-level event

Table 9 ILED with hierarchical bias

	ILED		
	$G = 10$	$G = 50$	$G = 100$
Level-1			
Training time (min)	4.46–4.88	5.78–6.44	6.24–6.88
Revisions	2–11	2–9	2–9
Hypothesis size	4–18	4–16	4–16
Precision (%)	100	100	100
Recall (%)	100	100	100
Level-2			
Training time (min)	8.76	9.14	9.86
Revisions	24	17	17
Hypothesis size	31	27	27
Precision (%)	100	100	100
Recall (%)	100	100	100
Level-3			
Training time (min)	5.78	6.14	6.78
Revisions	6	5	5
Hypothesis size	13	10	10
Precision (%)	100	100	100
Recall (%)	100	100	100

(*drivingStyle*) is relatively complex, in contrast to the simpler level-3 definition, for which training times are comparable to the ones of level-1 events.

The largest parts of training times were dedicated to checking an already correct definition against the part of the dataset that had not been processed yet. That is, for all target events, ILED converged to a complete definition relatively quickly, i.e. in approximately 1.5–3 min after the initiation of the learning process. From that point on, the extra time was spent on testing the hypothesis against the new incoming data.

Window granularity slightly affects the produced hypothesis for all target high-level events. Indeed, the definitions constructed with windows of 10 examples are slightly larger than the ones constructed with larger window sizes of 50 and 100 examples. Notably, the definitions constructed with windows of granularity 50 and 100, were found concise, meaningful and very close to the actual hand-crafted rules that were utilized in PRONTO.

6 Related work

A thorough review of the drawbacks of state-of-the-art ILP systems with respect to non-monotonic domains, as well as the deficiencies of existing approaches to learning Event Calculus programs can be found in Ray (2009), Otero (2001, 2003) and Sakama (2005, 2001). The main obstacle, common to many learners which combine ILP with some form of abduction, like PROGOL5 (Muggleton and Bryant 2000), ALECTO (Moyle 2003), HAIL (Ray et al. 2003) and IMPARO (Kimber et al. 2009), is that they cannot perform abduction through negation and are thus essentially limited to Observational Predicate Learning.

TAL (Corapi et al. 2010) is a top–down non-monotonic learner which is able to solve the same class of problems as XHAIL. It obtains a top theory by appropriately mapping an ILP

problem to a corresponding ALP instance, so that solutions to the latter may be translated to solutions for the initial ILP problem. Recently, the main ideas behind TAL were employed in the ASPAL system (Corapi et al. 2012), an inductive learner which relies on Answer Set Programming as a unifying abductive–inductive framework.

In Athakravi et al. (2013) the methodology behind TAL and ASPAL have been ported into a learner that constructs hypotheses progressively, towards more scalable learning. To address the fact that ASPALs top theory grows exponentially with the length of its clauses, RASPAL, the system proposed in Athakravi et al. (2013), imposes bounds on the length of the top theory. Partial hypotheses of specified clause length are iteratively obtained in a refinement loop. At each iteration of this loop, the hypothesis obtained from the previous refinement step is further refined by dropping or adding literals or clauses, using theory revision as described in Corapi et al. (2008). The process continues until a complete and consistent hypothesis is obtained. The main difference between RASPAL and our approach is that in order to ensure soundness, RASPAL has to process all examples simultaneously. At each iteration of its refinement loop, all examples are taken into account repeatedly, in order to ensure that the revisions account for all them. Therefore, the grounding bottleneck that RASPAL faces is expected to persist in domains that involve large volumes of sequential data, typical of temporal applications, as the ones that we address in this work. This is because even by imposing a small initial maximum clause length to RASPAL, in order to constrain the search space, with a sufficient amount of data the resulting ground program will still be intractable, if the data is processed simultaneously. In contrast, ILED is able to break the dataset in smaller data windows and process them in isolation, while ensuring soundness. By properly restricting window size, so that the unit cost of learning/revising from a single window is acceptable, ILED scales to large volumes of data, since the cost of theory revision grows as a linear function of the number of example windows in the historical memory.

The combination of ILP with ALP has recently been applied to *meta-interpretive learning* (MIL), a learning framework where the goal is to obtain hypotheses in the presence of a meta-interpreter. The latter is a higher-order program, hypothesizing about predicates or even rules of the domain. Given such background knowledge and a set of examples, MIL uses abduction w.r.t. the meta-interpreter to construct first-order hypotheses. MIL can be realized both in Prolog and in Answer Set Programming, and it has been implemented in the METAGOL system (Muggleton et al. 2014). MIL is an elegant framework, able to address difficult problems like predicate invention and mutually recursive programs. However, it has a number of important drawbacks. First, its expressivity is limited, as MIL is currently restricted to *dyadic Datalog*, i.e. Datalog where the arity of each predicate is at most two. Second, given the increased computational complexity of higher-order reasoning, scaling to large volumes of data is a potential bottleneck for MIL.

CLINT (De Raedt and Bruynooghe 1994) is a seminal abductive–inductive theory revision system. It employs two revision operators: A generalization operator that adds new clauses/facts to the theory and a specialization operator, that retracts incorrect clauses from the theory. To generate new clauses, CLINT uses "starting clauses", i.e. variabilized, most-specific clauses that cover a positive example, which are then maximally generalized to obtain a good hypothesis clause. CLINT also uses abduction in order to explain some examples, by deriving new ground facts which are simply added to the theory. Abduction and induction are independent and complementary, i.e. one is used when the other fails to cover an example, contrary to ILED, where the two processes are tightly coupled, allowing to handle non-OPL. Additionally, CLINT is restricted to Horn logic.

In Duboc et al. (2009), the theory revision system FORTE (Richards and Mooney 1995) is enhanced by PROGOL's bottom set construction routine and mode declarations, towards a

more efficient refinement operator. In order to refine a clause *C*, FORTE_MBC (the resulting system), uses mode declarations and inverse entailment to construct a bottom clause from a positive example covered by *C*. It then searches for antecedents within the bottom clause. As in the case of ILED, the constrained search space results in a more efficient clause refinement process. However FORTE_MBC (like FORTE itself) learns Horn theories and does not support non-Observational Predicate Learning, thus it cannot be used for the revision of Event Calculus programs. In addition, it cannot operate on an empty hypothesis (i.e. it cannot induce a hypothesis from scratch).

INTHELEX (Esposito et al. 2000) learns/revises Datalog theories and has been used in the study of several aspects of incremental learning, such as order effects (Di Mauro et al. 2004, 2005) and concept drift (Esposito et al. 2004). In Biba et al. (2006) the authors present an approach towards scaling INTHELEX by associating clauses in the theory at hand with examples they cover, via a relational schema. Thus, when a clause is refined, only the examples that were previously covered by this clause are checked. Similarly, when a clause is generalized, only the negative examples are checked again. The scalable version of INTHELEX presented in Biba et al. (2006) maintains alternative versions of the hypothesis at each step, allowing it to backtrack to previous states. In addition, it keeps in memory several statistics related to the examples that the system has already seen, such as the number of refinements that each example has caused, a "refinement history" of each clause, etc.

Several limitations make INTHELEX inappropriate for inducing/revising Event Calculus programs. First, the restriction of its input language to Datalog limits its applicability to richer, relational event domains. For instance, complex relations between entities cannot be easily expressed in INTHELEX. Second, the use of background knowledge is limited, excluding for instance auxiliary clauses that may be used for spatio-temporal reasoning during learning time. Third, although INTHELEX uses abduction for the completion of imperfect input data, it relies on Observational Predicate Learning, meaning that it is not able to reason with predicates which are not directly observable in the examples.

7 Conclusions

We presented an incremental ILP system, ILED, for constructing event recognition knowledge bases in the form of Event Calculus theories. ILED combines techniques from non-monotonic ILP and in particular, the XHAIL algorithm, with theory revision. It acquires an initial hypothesis from the first available piece of data, and revises this hypothesis as new data arrive. Revisions account for all accumulated experience. The main contribution of ILED is that it scales-up XHAIL to large volumes of sequential data with a time-like structure, typical of event-based applications. By means of a compressive memory structure that supports clause refinement, ILED has a scalable, single-pass revision strategy, thanks to which the cost of theory revision grows as a tractable function of the perceived experience. In this work, ILED was evaluated on an activity recognition application and a transport management application. The results indicate that ILED is significantly more efficient than XHAIL, without compromising the quality of the generated hypothesis in terms of predictive accuracy and hypothesis size. Moreover, ILED scales adequately to large data volumes which XHAIL cannot handle. Future work concerns mechanisms for handling noise and concept drift.

Appendix 1: Notions from (Inductive) Logic Programming

Interpretations and models (Gelfond and Lifschitz 1988). Given a logic program Π a Herbrand interpretation I is a subset of the set of all possible groundings of Π. I satisfies a literal a (resp. not a) iff $a \in I$ (resp. $a \notin I$). I satisfies a set of ground atoms iff it satisfies each one of them and it satisfies a ground clause iff it satisfies the head, or does not satisfy at least one body literal. I is a Herbrand model of Π iff it satisfies every ground instance of every clause in Π and it is a minimal model iff no strict subset of I is a model of Π. I is a *stable model* of Π iff it is a minimal model of the Horn program that results from the ground instances of Π after the removal of all clauses with a negated literal not satisfied by I, and all negative literals from the remaining clauses.

Mode Declarations and mode language (Muggleton 1995). A *mode declaration* is an atom of the form *modeh(s)* or *modeb(s)*, where s is called a schema. A schema s is a ground literal containing *placemarkers*. A placemarker is either $+type$ (input) $-type$ (output) or $\#type$ (ground), where *type* is a constant. A set M of mode declarations defines a language $\mathcal{L}(M)$. A clause C is in $\mathcal{L}(M)$ iff its head atom (respectively each of its body literals) is constructed from the schema s in a *modeh(s)* atom (resp. in a *modeb(s)* atom) in M by: (a) replacing an output placemarker by a new variable; (b) replacing an input placemarker by a variable that appears in the head atom, or in a previous body literal; (c) replacing a ground placemarker by a ground term. A hypothesis H is in $\mathcal{L}(M)$ iff $C \in \mathcal{L}(M)$ for each $C \in H$.

Appendix 2: ILED's high-level strategy and Proofs of Propositions

---

Algorithm 3 `iled`(SDEC, M, H_n, w_n) (**ILED's High-Level Strategy**)
Input: *The axioms of* SDEC, *mode declarations* M, *a hypothesis* H_n *such that* SDEC $\cup H_n \models$ $\mathcal{E}$ *and an example window* w_n.
Output: *A hypothesis* H_{n+1} *such that* SDEC $\cup H_{n+1} \models \mathcal{E} \cup w_n$

1: **if** SDEC $\cup H_n \nvDash w_n$ **then**
2:      **let** $K_v^{w_n}$ be a (variabilized) Kernel Set of w_n
3:      **let** $\langle RetainedClauses, RefinedClauses, NewClauses \rangle \leftarrow$ revise(SDEC, H_n, $K_v^{w_n}$, w_n)
4:      **let** $H' \leftarrow H_{keep} \cup RefinedClauses \cup NewClauses$
5:      **if** $NewClauses \neq \emptyset$ **then**
6:          **for all** $w_i \in \mathcal{E}$, $0 \le i \le n-1$ **do**
7:              **if** SDEC $\cup H' \nvDash w_i$ **then**
8:                  **let** $\langle RetainedClauses, RefinedClauses, \emptyset \rangle \leftarrow$ revise(SDEC, H', $\emptyset$, w_i)
9:                  **let** $H' \leftarrow RetainedClauses \cup RefinedClauses$
10:     **let** $H_{n+1} \leftarrow H'$
11: **else**
12:      **let** $H_{n+1} \leftarrow H_n$
13: **let** $\mathcal{E} \leftarrow \mathcal{E} \cup w_n$
14: **Return** H_{n+1}

Proof of Proposition 1 We first show that $cov_{\mathcal{E}}(C.supp) = cov_{\mathcal{E}}(C)$. For the inclusion $cov_{\mathcal{E}}(C.supp) \subseteq cov_{\mathcal{E}}(C)$, assume that $e \in cov_{\mathcal{E}}(C.supp)$, i.e. e is covered by a $D \in C.supp$. But C θ-subsumes D, therefore $e \in cov_{\mathcal{E}}(C)$. For the inverse inclusion, assume that $e \in cov_{\mathcal{E}}(C)$ and let D be the most-specific clause of $\mathcal{L}(M)$, such that $e \in cov_{\mathcal{E}}(D)$ $(\star)$ (observe that if no such D exists, with $D \neq C$, then C itself is the most-specific clause

with the required property). Then by definition, $D \in C.supp$ and from ($\star$) we have that $e \in cov_{\mathcal{E}}(C.supp)$, establishing the inclusion $cov_{\mathcal{E}}(C) \subseteq cov_{\mathcal{E}}(C.supp)$.

The fact that $C.supp$ is the most-specific program of $\mathcal{L}(M)$ with this property follows immediately from Definition 3, since each clause in $\mathcal{L}_i(C.supp)$ is most-specific in $\mathcal{L}(M)$ with the property of covering at least one example from $cov_{\mathcal{E}}(C)$.   □

Proposition 2 *Let $H_n \in \mathcal{L}(M)$ be as in the Incremental Learning setting (Definition 1), i.e.* $\mathsf{SDEC} \cup H_n \vDash \mathcal{E}$, *and w_n be an example window. Assume also that there exists a hypothesis $H_{n+1} \in \mathcal{L}(M)$, such that $\mathsf{SDEC} \cup H_{n+1} \vDash \mathcal{E} \cup w_n$, and that a clause $C \in H_n$ is revisable w.r.t. window w_n. Then $C.supp$ contains a refinement R_C of C, which is preservable w.r.t. w_n.*

Proof Assume, towards contradiction, that each refinement R_C of C, contained in $C.supp$ is revisable w.r.t. w_n. It then follows that $C.supp$ itself is revisable w.r.t. w_n, i.e. it either covers some negative examples, or it disproves some positive examples in w_n. Let $e_1 \in w_n$ be such an example that $C.supp$ fails to satisfy, and assume for simplicity that a single clause $C_s \in C.supp$ is responsible for that. By definition, C_s covers at least one positive example e_2 from $\mathcal{E}$ and furthermore, it is a most-specific clause, within $\mathcal{L}_i(M)$, with that property. It then follows that e_1 and e_2 cannot both be accounted for, under the given language bias $\mathcal{L}(M)$, i.e. there exists no hypothesis $H_{n+1} \in \mathcal{L}(M)$ such that $\mathsf{SDEC} \cup H_{n+1} \vDash \mathcal{E} \cup w_n$, which contradicts our assumption. Hence $C.supp$ is preservable w.r.t. w_n and it thus contains a refinement R_C of C, which is preservable w.r.t. w_n.   □

Proposition 3 (Soundness and Single-pass Theory Revision) *Assume the incremental learning setting described in Definition 1. ILED requires at most one pass over $\mathcal{E}$ to compute H_{n+1} from H_n.*

Proof For simplicity and without loss of generality, we assume that when a new example window w_n arrives, ILED revises H_n by (a) refining an single clause $C \in H_n$ or (b) adding a new clause C'.

In case (a), clause C is replaced by a refinement R_C such that $C \preceq R_C \preceq C.supp$. By property (iii) of the support set (see Proposition 1), R_C covers all positive examples that C covers in $\mathcal{E}$, hence for the hypothesis $H_{n+1} = (H_n \smallsetminus C) \cup R_C$ it holds that $\mathsf{SDEC} \cup H_{n+1} \vDash \mathcal{E}$ and furthermore $\mathsf{SDEC} \cup H_{n+1} \vDash w_n$. Hence $\mathsf{SDEC} \cup H_{n+1} \vDash \mathcal{E} \cup w_n$, from which soundness for H_{n+1} follows. In this case H_{n+1} is constructed from H_n in a single step, i.e. by reasoning within w_n without re-seeing other windows from $\mathcal{E}$.

In case (b), H_n is revised w.r.t. w_n to a hypothesis $H'_n = H_n \cup C'$, where C' is a new clause that results from the generalization of a Kernel Set of w_n. In response to the new clause addition, each window in $\mathcal{E}$ must be checked and C' must be refined if necessary. Let $\mathcal{E}_{tested}$ denote the fragment of $\mathcal{E}$ that has been tested at each point in time. Initially, i.e. once C' is generated from w_n, it holds that $\mathcal{E}_{tested} = w_n$. At each window that is tested, clause C' may (i) remain intact, (ii) be refined, or (iii) one of its refinements may be further refined. Assume that w_k, $k < n$ is the first window where the new clause C' must be refined. At this point, $\mathcal{E}_{tested} = \{w_i \in \mathcal{E} \mid k \leq i \leq n\}$, and it holds that C' is preservable in $\mathcal{E}_{tested}$, since C' has not yet been refined. In w_k, clause C' is replaced by a refinement $R_{C'}$ such that $C' \preceq R_{C'} \preceq C'.supp$. $R_{C'}$ is preservable in $\mathcal{E}_{tested}$, since it is a refinement of a preservable clause, and furthermore, it covers all positive examples that C' covers in w_n, by means of the properties of the support set. Hence the hypothesis $H''_n = (H'_n \smallsetminus C') \cup R_{C'}$ is complete & consistent w.r.t. $\mathcal{E}_{tested}$. The same argument shows that if $R_{C'}$ is further refined later on (case (iii) above), the resulting hypothesis remains complete an consistent w.r.t. $\mathcal{E}_{tested}$. Hence,

when all windows have been tested, i.e. when $\mathcal{E}_{tested} = \mathcal{E}$, the resulting hypothesis H_{n+1} is complete & consistent w.r.t. $\mathcal{E} \cup w_n$ and furthermore, each window in $\mathcal{E}$ has been re-seen exactly once, thus H_{n+1} is computed with a single pass over $\mathcal{E}$. □

References

Ade, H., & Denecker, M. (1995). AILP: Abductive inductive logic programming. In *Proceedings of the international joint conference on artificial intelligence (IJCAI)*.

Alrajeh, D., Kramer, J., Russo, A., & Uchitel, S. (2009). Learning operational requirements from goal models. In *Proceedings of the 31st international conference on software engineering* (pp. 265–275). IEEE Computer Society.

Alrajeh, D., Kramer, J., Russo, A., & Uchitel, S. (2010). Deriving non-zeno behaviour models from goal models using ILP. *Formal Aspects of Computing*, *22*(3–4), 217–241.

Alrajeh, D., Kramer, J., Russo, A., & Uchitel, S. (2011). An inductive approach for modal transition system refinement. In *Technical communications of the international conference of logic programming ICLP* (pp. 106–116). Citeseer.

Alrajeh, D., Kramer, J., Russo, A., & Uchitel, S. (2012). Learning from vacuously satisfiable scenario-based specifications. In *Proceedings of the international conference on fundamental approaches to software engineering (FASE)*.

Artikis, A., Skarlatidis, A., & Paliouras, G. (2010). Behaviour recognition from video content: A logic programming approach. *International Journal on Artificial Intelligence Tools*, *19*(2), 193–209.

Artikis, A., Skarlatidis, A., Portet, F., & Paliouras, G. (2012). Logic-based event recognition. *Knowledge Engineering Review*, *27*(04), 469–506.

Artikis, A., Sergot, M., & Paliouras, G. (2015). An event calculus for event recognition. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, *27*(4), 895–908.

Athakravi, D., Corapi, D., Broda, K., & Russo, A. (2013). Learning through hypothesis refinement using answer set programming. In *Proceedings of the 23rd international conference of inductive logic programming (ILP)*.

Badea, L. (2001). A refinement operator for theories. In *Proceedings of the international conference on inductive logic programming (ILP)*.

Biba, M., Basile, T. M. A., Ferilli, S., & Esposito, F. (2006). Improving scalability in ILP incremental systems. In *Proceedings of CILC 2006-Italian conference on computational logic, Bari, Italy*, pp. 26–27.

Bragaglia, S. & Ray, O. (2014). Nonmonotonic learning in large biological networks. In *Proceedings of the international conference on inductive logic programming (ILP)*.

Cattafi, M., Lamma, E., Riguzzi, F., & Storari, S. (2010). Incremental declarative process mining. *Smart Information and Knowledge Management*, *260*, 103–127.

Cervesato, I., & Montanari, A. (2000). A calculus of macro-events: Progress report. In *Proceedings of the international workshop on temporal representation and reasoning (TIME)*. IEEE.

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

Corapi, D., Ray, O., Russo, A., Bandara, A., & Lupu, E. (2008). Learning rules from user behaviour. In *Second international workshop on the induction of process models*.

Corapi, D., Russo, A., & Lupu, E. (2010). Inductive logic programming as abductive search. In *Technical communications of the international conference on logic programming (ICLP)*.

Corapi, D., Russo, A., & Lupu, E. (2012). Inductive logic programming in answer set programming. In *Proceedings of international conference on inductive logic programming (ILP)*. Springer.

De Raedt, L., & Bruynooghe, M. (1994). Interactive theory revision. In *Machine learning: A multistrategy approach*, pp. 239–263.

Denecker, M., & Kakas, A. (2002). Abduction in logic programming. In *Computational logic: Logic programming and beyond*, pp. 402–436.

Di Mauro, N., Esposito, F., Ferilli, S., & Basile, T. M. A. (2004). A backtracking strategy for order-independent incremental learning. In *Proceedings of the European conference on artificial intelligence (ECAI)*.

Di Mauro, N., Esposito, F., Ferilli, S., & Basile, T. M. (2005). Avoiding order effects in incremental learning. In *AIIA 2005: Advances in artificial intelligence*, pp. 110–121.

Dietterich, T. G., Domingos, P., Getoor, L., Muggleton, S., & Tadepalli, P. (2008). Structured machine learning: The next ten years. *Machine Learning*, *73*, 3–23.

Duboc, A. L., Paes, A., & Zaverucha, G. (2009). Using the bottom clause and mode declarations in FOL theory revision from examples. *Machine Learning*, *76*(1), 73–107.

Eshghi, K., & Kowalski, R. (1989). Abduction compared with negation by failure. In *Proceedings of the 6th international conference on logic programming*.

Esposito, F., Semeraro, G., Fanizzi, N., & Ferilli, S. (2000). Multistrategy theory revision: Induction and abduction in inthelex. *Machine Learning*, *28*(1–2), 133–156.

Esposito, F., Ferilli, S., Fanizzi, N., Basile, T. M. A., & Di Mauro, N. (2004). Incremental learning and concept drift in inthelex. *Intelligent Data Analysis*, *8*(3), 213–237.

Etzion, O., & Niblett, P. (2010). *Event processing in action*. Greenwich: Manning Publications Co.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *6*(3), 1–238.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In *International conference on logic programming*, pp. 1070–1080.

Kakas, A., & Mancarella, P. (1990). Generalised stable models: A semantics for abduction. In Ninth European conference on artificial intelligence (ECAI-90), pp. 385–391.

Kakas, A., Kowalski, R., & Toni, F. (1993). Abductive logic programming. *Journal of Logic and Computation*, *2*, 719–770.

Kimber, T., Broda, K., & Russo, A. (2009). Induction on failure: Learning connected horn theories. In *Logic programming and nonmonotonic reasoning*, pp. 169–181.

Kowalski, R., & Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, *4*(1), 6796.

Kuzelka, O., & Zelezny, F. (2008). A restarted strategy for efficient subsumption testing. *Fundamenta Informaticae*, *89*(1), 95–109.

Langley, P. (1995). *Learning in humans and machines: Towards an interdisciplinary science, chapter order effects in incremental learning*. Amsterdam: Elsevier.

Lavrač, N., & Džeroski, S. (1993). *Inductive logic programming: Techniques and applications*. London: Routledge.

Li, H.-F., & Lee, S.-Y. (2009). Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Systems with Applications*, *36*(2), 1466–1477.

Li, H.-F., Lee, S.-Y., & Shan, M.-K. (2004). An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proceedings of first international workshop on knowledge discovery in data streams*.

List, T., Bins, J., Vazquez, J., & Fisher, R. B. (2005). Performance evaluating the evaluator. In *2nd joint IEEE international workshop on visual surveillance and performance evaluation of tracking and surveillance* (pp. 129–136). IEEE.

Lloyd, J. (1987). *Foundations of logic programming*. Berlin: Springer.

Luckham, D. (2001). *The power of events: An introduction to complex event processing in distributed enterprise systems*. Boston: Addison-Wesley Longman Publishing Co., Inc.

Luckham, D., & Schulte, R. (2008). *Event processing glossary, version 1.1*. Trento: Event Processing Technical Society.

Maloberti, J., & Sebag, M. (2004). Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, *55*(2), 137–174.

Mitchell, T. (1979). *Version spaces: An approach to concept learning*. PhD thesis, AAI7917262.

Moyle, S. (2003). *An investigation into theory completion techniques in inductive logic*. PhD thesis, University of Oxford.

Mueller, E. (2006). *Commonsense reasoning*. Burlington: Morgan Kaufmann.

Mueller, E. T. (2008). Event calculus. *Foundations of Artificial Intelligence*, *3*, 671–708.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing*, *13*(3&4), 245–286.

Muggleton, S., & Bryant, C. (2000). Theory completion using inverse entailment. In *International conference on inductive logic programming*, pp. 130–146.

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, *19*, 629–679.

Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K., et al. (2012). ILP turns 20. *Machine Learning*, *86*(1), 3–23.

Muggleton, S. H., Lin, D., Pahlavi, N., & Tamaddoni-Nezhad, A. (2014). Meta-interpretive learning: Application to grammatical inference. *Machine Learning*, *94*(1), 25–49.

Otero, R. P. (2001). Induction of stable models. *Inductive Logic Programming*, *2157*, 193–205.

Otero, R. P. (2003). Induction of the effects of actions by monotonic methods. *Inductive Logic Programming*, *2835*, 299–310.

Paschke, A. (2005). *ECA-RuleML: An approach combining ECA rules with temporal interval-based KR event logics and transactional update logics*. Technical report, Technische Universitat Munchen.

Ray, O. (2006). Using abduction for induction of normal logic programs. In *ECAI'06 workshop on abduction and induction in articial intelligence and scientic modelling*.

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

Ray, O., Broda, K., & Russo, A. (2003). Hybrid abductive inductive learning: A generalisation of progol. In *Proceedings of the international conference in inductive logic programming (ILP)*.

Richards, B., & Mooney, R. (1995). Automated refinement of first-order horn clause domain theories. *Machine Learning*, *19*(2), 95–131.

Sakama, C. (2000). Inverse entailment in nonmonotonic logic programs. In *Proceedings of the international conference on inductive logic programming (ILP)*.

Sakama, C. (2001). Nonmonotomic inductive logic programming. In *Logic programming and nonmotonic reasoning* (pp. 62–80). Springer.

Sakama, C. (2005). Induction from answer sets in nonmonotonic logic programs. *ACM Transactions on Computational Logic*, *6*(2), 203231.

Santos, J., & Muggleton, S. (2010). Subsumer: A prolog theta-subsumption engine. In *Technical communications of the 26th international conference on logic programming*.

Sloman, M., & Lupu, E. (2010). Engineering policy-based ubiquitous systems. *The Computer Journal*, *53*(5), 1113–1127.

Wrobel, S. (1996). First order theory refinement. In L. De Raedt (Ed.), *Advances in inductive logic programming* (pp. 14–33). Citeseer.