Contents lists available at ScienceDirect

Journal of Applied Logic

www.elsevier.com/locate/jal

Using abduction and induction for operational requirements elaboration

D. Alrajeh [a,*], O. Ray [b], A. Russo [a], S. Uchitel [a,c]

[a] *Imperial College London, Department of Computing, 180 Queen's Gate, London SW7 2AZ, United Kingdom*
[b] *University of Bristol, Department of Computer Science, Woodland Road, Bristol BS8 1UB, United Kingdom*
[c] *University of Buenos Aires, Departamento de Computación, Ciudad Universitaria Intendente Güiraldes 2160, Buenos Aires, (C1428EGA) Argentina*

A R T I C L E   I N F O

Article history:
Received 18 December 2007
Received in revised form 18 December 2007
Accepted 19 October 2008
Available online 6 November 2008

Keywords:
Inductive logic programming
Abductive reasoning
Goal-oriented requirements engineering
Scenario-based specification
Linear temporal logic
Event Calculus

A B S T R A C T

Requirements Engineering involves the elicitation of high-level stakeholder goals and their refinement into operational system requirements. A key difficulty is that stakeholders typically convey their goals indirectly through intuitive narrative-style scenarios of desirable and undesirable system behaviour, whereas goal refinement methods usually require goals to be expressed declaratively using, for instance, a temporal logic. In actual software engineering practice, the extraction of formal requirements from scenario-based descriptions is a tedious and error-prone process that would benefit from automated tool support. This paper presents an Inductive Logic Programming method for inferring operational requirements from a set of example scenarios and an initial but incomplete requirements specification. The approach is based on translating the specification and the scenarios into an event-based logic programming formalism and using a non-monotonic reasoning system, called eXtended Hybrid Abductive Inductive Learning, to automatically infer a set of event pre-conditions and trigger-conditions that cover all desirable scenarios and reject all undesirable ones. This learning task is a novel application of logic programming to requirements engineering that also demonstrates the utility of non-monotonic learning capturing pre-conditions and trigger-conditions.

1. Introduction

Requirements Engineering is an integral part of the software engineering life-cycle. It is concerned with the elicitation, elaboration, specification, analysis and documentation of goals and requirements relating to an envisaged system. It aims to bridge the gap between high-level goals conveyed by stakeholders (e.g. users, developers and decision makers), and low-level operational system requirements. Some methods have been developed to support the elicitation and elaboration of such requirements [11,12] but these focus on the refinement of high-level goals declaratively expressed in a temporal logic [15]. The use of a temporal formalism enables the deployment of automated analysis and refinement tools, but is not directly accessible to most stakeholders with a less technical background. In practice, stakeholders prefer to convey their goals through more intuitive narrative-style scenarios of desirable and undesirable system behaviour [29] rather than temporal assertions. Because scenarios are inherently *partial* descriptions about specific system behaviours, they leave requirements implicitly defined. It is therefore necessary to synthesise declarative specifications of operational requirements that admit the desired behaviours while rejecting the undesired ones. At present, the elicitation of declarative requirements from scenario-based descriptions is a tedious and error-prone process that relies on the manual efforts of an experienced engineer and would benefit from automated tool support.

[*] Corresponding author.
E-mail addresses: da04@doc.ic.ac.uk (D. Alrajeh), oray@cs.bris.ac.uk (O. Ray), ar3@doc.ic.ac.uk (A. Russo), s.uchitel@doc.ic.ac.uk (S. Uchitel).

1570-8683/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.jal.2008.10.002

This paper provides an Inductive Logic Programming (ILP) approach [21] for *extracting* operational requirements from example scenarios and an initial but incomplete system specification. The approach presented here extends the one introduced in [1], which used a non-monotonic ILP system to learn a particular type of requirements, called event *pre-conditions* [12], from example scenarios and a partial system specification. In this paper, we show how this technique can be extended to learn another class of operational requirements called event *trigger-conditions* [12]. Intuitively, pre-conditions state that a certain event *should not* happen in some situations, while *trigger-conditions* state that an event *must* happen in some other situations.

The scenarios from which operational requirements are extracted represent examples of desirable and undesirable system behaviour. Scenarios are sequences of events that can either occur in the environment or be performed by the system. Desirable behaviour is represented by *positive scenarios*, which are sequences of events that are consistent with the system specification. Undesirable behaviour is represented by *negative scenarios*, which are sequences of events that are not consistent with the system specification. A partial system specification is also given as input to our approach in order to represent any initial but incomplete knowledge of the envisioned system and its environment.

The task is to complete the specification by learning a set of missing event pre-conditions and trigger-conditions that cover all of the positive scenarios, but none of the negative ones. To achieve this, we show how the partial specification and scenarios can be expressed in Linear Temporal Logic (LTL) [15] and automatically transformed into an ILP representation based on the Event Calculus (EC) framework [7,16] by means of a sound translation process. Because this representation makes essential use of negation in formalising the effects and non-effects of actions, the resulting problem is inherently non-monotonic. The main challenge is to learn general pre-conditions and trigger-conditions expressed not in terms of specific sequences of actions, but expressed in terms of the relevant effects of those actions.

We show that, under the stable model semantics [5] for logic programs with negation, the stable models of the transformed program correspond to the temporal models of the original specification. We then use the non-monotonic learning system called eXtended Hybrid Abductive Inductive Learning (XHAIL) [23,24] to generalise the scenarios with respect to the initial requirements specification. The abuctive component of XHAIL generates a ground unit explanation of the example scenario. This preliminary explanation is then generalised by XHAIL's inductive component to produce the final hypothesis. In this way, XHAIL integrates assumption-based and generalisation-based inference within a coherent non-monotonic learning framework. The language bias of XHAIL is defined so as to allow the inference of pre-conditions and trigger-conditions. We show that once the EC clauses learnt by XHAIL are translated back to LTL formulae, they provide a *correct extension* of the given partial system specifications with respect to the given example scenarios.

The paper is organized as follows. Section 2 presents the relevant background material on Inductive Logic Programming, XHAIL, Linear Temporal Logic, and the Event Calculus. Section 3 describes the main features of our approach and presents some results on the soundness of the translation process of LTL requirements specifications into EC logic programs and on the correspondence between LTL and EC extensions of the initial specification. Section 4 provides an illustrative case study involving a Mine Pump controller, where event pre-conditions and trigger-conditions are simultaneously learned using the XHAIL system. A summary and some remarks about related and future work conclude the paper.

2. Background

This section introduces the necessary background material. After a summary of general notation and terminology, we describe the two logic-based formalisms used in our approach: Linear Temporal Logic, for expressing requirements specifications, and the Event Calculus for reasoning logically about action and change.

2.1. Notation and terminology

A term is either a constant, a variable or a compound term $f(t_1,\ldots,t_n)$ where f is a function symbol and t_i is a term. A *literal* is an atomic formula $p(t_1,\ldots,t_n)$, also called an atom or a positive literal, or its negation, *not* $p(t_1,\ldots,t_m)$, also called a negative literal, where p is a predicate symbol, t_i is a term and *not* is the negation as failure operator. We use (*not*) p to refer to either the atom p *or* the negative literal *not* p of that atom. A *clause* is an expression of the form $\phi \leftarrow \psi_1,\ldots,\psi_n$ where ϕ is an atom (called the head atom) and ψ_i is a literal (called a body literal). A clause is *ground* if it contains no variables. A clause is *definite* if all of its body literals are positive. The *empty clause* is denoted $\square$ and represents the truth value *false*. A *goal clause* is a clause $(\leftarrow \psi_1,\ldots,\psi_n)$ with an empty head. A *logic program* is set of clauses. A definite logic program is a program in which all clauses are definite. A normal logic program is one in which the clauses are of the form $A \leftarrow B_1,\ldots,B_n$, *not* $C_1,\ldots,$ *not* C_m where A is the *head atom*, B_i are *positive body literals*, and *not* C_j are *negative body literals*.

A (Herbrand) *model* I of a normal logic program, Π, is a set of ground atoms such that, for each ground instance G of a clause in Π, I satisfies the head of G whenever it satisfies the body. A model I is *minimal* if it does not strictly include any other model. Definite programs always have a unique minimal model. Normal programs may have instead one, none, or several minimal models. It is *customary* to identify a certain subset of these models, called *stable models*, as the possible meanings of the program. Given a normal logic program Π, the *reduct* of Π with respect to I, denoted Π^I, is the program obtained from the ground instances of Π by (a) removing all clauses with a negative literal *not* a in its body where $a \in I$

and (b) removing all negative literals from the bodies of the remaining clauses. If I is the least Herbrand model of Π^I then I is said to be a stable model of Π as formalised in the definition below.

Definition 2.1. A model I of Π is a *stable model* if I is the least Herbrand model of Π^I where Π^I is the definite program $\Pi^I = \{A \leftarrow B_1, \ldots, B_n \mid A \leftarrow B_1, \ldots, B_n,\ \mathit{not}\ C_1, \ldots, \mathit{not}\ C_n$ is the ground instance of a clause in Π and I does not satisfy any of the $C_i\}$.

The definitions of entailment under stable model semantics is given below.

Definition 2.2. A program Π *entails* an expression E (under the credulous stable model semantics), denoted $\Pi \models E$, iff E is satisfied in at least one stable model of Π.

2.2. Inductive Logic Programming and XHAIL

ILP is concerned with the computation of hypotheses H that generalise a set of (positive and negative) examples E with respect to a prior background theory B. In this paper we consider the case when B and H are normal logic programs, E is a set of ground literals (with positive and negative literals representing positive and negative examples, respectively), and H satisfies the condition $B \cup H \models E$ under the stable model semantics. In other words, the examples E must be satisfied in a stable model of $B \cup H$. As formalised in the definition below, it is usual to further restrict the clauses in H to a set of clauses HS called a *hypothesis space*.

Definition 2.3. Given a normal logic program B, a set of ground literals E, and a set clauses HS, the task of ILP is to find a normal logic program $H \subseteq \mathrm{HS}$, consistent with B such that $B \cup H \models E$. In this case, H is called an inductive generalisation of E w.r.t. B and HS.

To implement our methodology, we used the XHAIL system [24], which is one of relatively few ILP systems designed for non-monotonic ILP. It is based on a three-phase Hybrid Abductive Inductive Learning (HAIL) approach [23] which operates by constructing and generalising a preliminary ground hypothesis K, called a *Kernel Set* of B and E. This notion can be regarded as a non-monotonic multi-clause generalisation of the *Bottom Set* concept used in several well-known monotonic ILP systems [19,20]. Like these monotonic ILP systems, XHAIL heavily exploits language and search bias when constructing and generalising a Kernel Set in order to bound the ILP hypothesis space.[1]

The XHAIL language and search bias mechanisms are based upon the tried-and-tested notions of compression and mode declarations as used, for example, in Progol [19]. The compression heuristic favours hypotheses containing the fewest number of literals and is motivated by the principle of Occam's razor (which, roughly speaking, means choose the simplest hypothesis that fits the data). Mode declarations provide a convenient mechanism for specifying which predicates may appear in the heads and bodies of hypothesis clauses and for controlling the placement and linking of constants and variables within those clauses.

As defined in [19], a mode declaration D is either a head declaration of the form $\mathit{modeh}(r, s)$ or a body declaration of the form $\mathit{modeb}(r, s)$, where r is an integer, called the recall, and s is a ground literal called the scheme, possibly containing so-called placemarker terms of the form $+t$, $-t$ and $\#t$, which must be replaced by input variables, output variables, and constants of type t, respectively. The recall is used to bound the number of atoms each mode declaration can contribute to a hypothesis clause. Where this is not important, an arbitrary recall is denoted by an asterisk $*$.

As explained in [24], XHAIL computes hypotheses using a non-monotonic abductive interpreter to implement the three stages of the HAIL approach. In the first phase, the head declarations are used to abduce the head atoms of the Kernel Set. In the second phase, the body atoms of the Kernel Set are computed as the successful instances of queries obtained from the body declaration schemas. In the third phase, the hypothesis is computed by searching for a compressive theory that subsumes the Kernel Set, is consistent with the background knowledge, covers the examples and falls within the hypothesis space.

2.3. Linear temporal logic

Several logic-based formalisms have been proposed for modeling event-based systems [6,9,25]. Among these, LTL [15] is widely used and is supported by analysis techniques and tools such as model checking [14]. The language of LTL includes a finite non-empty set of Boolean propositions P, Boolean connectives $\neg, \wedge$ and $\rightarrow$ and temporal operators $\bigcirc$ (next), $\Box$ (always), and U (strong until). A well-formed LTL formula is constructed such that:

- *ff* and *tt*, representing truth and falsity respectively, are formulae

[1] Further details about XHAIL can be found in Oliver Ray's paper *Non-monotonic Abductive Inductive Learning* elsewhere in this issue.

- an atomic proposition p is a formula
- if ϕ and ψ are formulae then so are: $\neg\phi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$, $\bigcirc\phi$, $\Box\phi$, and $\phi\ U\ \psi$.

Other formulae are introduced as abbreviations: $\phi \vee \psi$ abbreviates $\neg(\neg\phi \wedge \neg\psi)$ and $\phi \Leftrightarrow \psi$ abbreviates $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. The formula $\Diamond\phi$ abbreviates $\neg\Box\neg\phi$, and $\phi\ W\ \psi$ abbreviates $(\Box\phi) \vee (\phi\ U\ \psi)$. We use $(\neg)p$ to refer to either the proposition p or the negation $\neg p$ of that proposition. Also, we use $\bigcirc^i$ to denote i consecutive applications of the $\bigcirc$ operator. We further assume P to be partitioned into two sets P_e and P_f denoting *event* and *fluent* propositions respectively. The semantics of an LTL formula can be defined with respect to a structure called a Labelled Transition System (LTS) [28].

Definition 2.4. A labelled transition system T is a tuple $(S, L, s_0, \mathcal{R})$ where S is a finite non-empty set of states, L is a finite non-empty set of labels, called the *alphabet*, s_0 is a subset of S, called the *set of initial states*, and $\mathcal{R} \subseteq S \times L \times S$ is a non-empty set of transition relations.

An *input* for an LTS T is a finite sequence of the form $\langle l_1, l_2, \ldots, l_n\rangle$ where $l_i \in L$. A *path* σ in T is a (possibly infinite) sequence of states and transitions (i.e. $s_0\mathcal{R}_{l_1}s_1\mathcal{R}_{l_2}s_2, \ldots$) such that for each $i \geqslant 0$ there is a transition relation $(s_i, l_{i+1}, s_{i+1}) \in \mathcal{R}$, with label l_{i+1}. An input $\langle l_1, l_2, \ldots, l_n\rangle$ is said to be *accepted* by T if there is a path $\sigma = s_0\mathcal{R}_{l_1}s_1, \ldots, s_{n-1}\mathcal{R}_{l_n}s_n$ in T where, for all $0 \leqslant i < n$, $(s_i, l_{i+1}, s_{i+1}) \in \mathcal{R}$. Note that, for a given path (resp. input), the index of a state (resp. event) is referred to as its *position* in that path (resp. input).

An LTL model M is a pair $\langle T, V\rangle$ consisting of an LTS, T, and a valuation function, V, that assigns to each fluent proposition an arbitrary set of pairs of path, in T, and positions in the path. The events, however, are not specified in V as their truth is implicitly determined by the transitions.

Definition 2.5. Given an LTL language with propositions $P = P_e \cup P_f$, an LTL *model* M is a pair $\langle T, V\rangle$ where T is an LTS with events P_e and V is a valuation function $V : P_f \Rightarrow 2^A$, where $A = \{(\sigma, i) \mid \sigma \textit{ is a path in } T \textit{ and } i \textit{ is a position in } \sigma\}$.

The satisfiability of an LTL formula in a model M is defined with respect to positions in a given path.

Definition 2.6. Given an LTL language with propositions $P = P_e \cup P_f$, an LTL model $M = \langle T, V\rangle$ and a path σ in T, the satisfaction of an LTL formula ϕ at a position $i \geqslant 0$ of the path σ, denoted $\sigma, i \models \phi$, is defined inductively as follows:

- $\sigma, 0 \not\models e$ for any event proposition $e \in P_e$
- $\sigma, i \models e$ iff e is the ith label in the path σ, where $e \in P_e$ and $i \geqslant 1$
- $\sigma, i \models f$ iff $(\sigma, i) \in V(f)$, where $f \in P_f$
- $\sigma, i \models \neg\phi$ iff $\sigma, i \not\models \phi$
- $\sigma, i \models \phi \wedge \psi$ iff $\sigma, i \models \phi$ and $\sigma, i \models \psi$
- $\sigma, i \models \bigcirc\phi$ iff $\sigma, i+1 \models \phi$
- $\sigma, i \models \Box\phi$ iff $\forall j \geqslant i.\ \sigma, j \models \phi$
- $\sigma, i \models \phi\ U\ \psi$ iff $\exists j \geqslant i.\ \sigma, j \models \psi$ and $\forall i \leqslant k < j.\ \sigma, k \models \phi$.

An LTL formula ϕ is said to be *satisfied in a path* σ iff it is satisfied at the initial position, i.e. $\sigma, 0 \models \phi$. Similarly, a set of formulae Γ (also called a theory) is said to be satisfied in a path σ if each formula $\psi \in \Gamma$ is satisfied in the path σ. Given the above notions of satisfiability, we can now give a formal definition of a model of an LTL theory.

Definition 2.7. Let $M = (T, V)$ be an LTL model and Γ be an LTL theory. M is said to be a *model* of Γ, denoted $\models_M \Gamma$, iff Γ is satisfied in every path σ in T.

Definition 2.8. Let Γ be a LTL theory, ϕ a LTL formula and $M = \langle T, V\rangle$ an LTL model. The formula ϕ is said to be *entailed* by Γ, written $\Gamma \models_M \phi$, iff ϕ is satisfied in each path σ in T that satisfies Γ.

2.4. Event calculus

The Event Calculus (EC) formalism [7] is particularly well-suited for reasoning about events and their effects over time [27] using logic programming techniques such as ILP. Its ontology is close enough to existing types of event-based requirements specifications to allow them to be mapped automatically into logical representations that can be used as a back-end to existing requirements engineering representational methods. In particular, an EC language includes three sorts of terms: *event* terms, *fluent* terms, and *time* terms. While time is represented by the non-negative integers $0, 1, 2, \ldots$, the event and fluent sorts are defined according to the domain being modelled. In this paper, we assume the EC language to include an additional sort called *scenarios*. The EC ontology includes the basic predicates *happens*, *initiates*, *terminates* and *holdsAt*. The atomic formula *happens*(e, t, s) indicates that event e occurs at time-point t in a given scenario s, while *initiates*(e, f, t, s) (resp. *terminates*(e, f, t, s)) means that, in a given scenario s, if event e were to occur at time t, it would cause fluent f to be true (resp. false) immediately afterwards. The predicate *holdsAt*(f, t, s) indicates that fluent f is true at time-point t in a given scenario s. The formalism includes also an auxiliary predicate, *clipped*, and three additional predicates *impossible*, *attempt* and *triggered*. The predicate *clipped*(t_1, f, t_2, s) means that, in a given scenario s, an event occurs which terminates f between times t_1 and t_2. The predicate *impossible*(e, t, s) means that in scenario s, event e cannot be performed at time t, the predicate *attempt*(e, t, s) states that in scenario s, event e may be attempted at time t, and *triggered*(e, t, s) indicates instead that in scenario s, event e has been triggered at time point t. Whereas the predicate *attempt* expresses the concept of a possible transition, the predicate *triggered* (resp. *impossible*) defines transitions that must (resp. cannot) occur. Their distinction is clearly captured by the translation function given in Section 3.2 for constructing EC programs from LTL requirement specifications.

An EC program includes domain-dependent axioms describing which actions *initiate* and *terminate* which fluents, which fluents are *initially* true, which events are *attempted*, and rules defining the *impossibility* and the *triggering* conditions of event occurrences. It also contains a narrative, which describes a course of events that may be attempted at specific time points and in specific scenarios, and a set of domain-independent axioms which govern the interactions between the EC predicates:

$$\mathit{clipped}(T_1, F, T_2, S) \leftarrow \mathit{happens}(E, T, S), \mathit{terminates}(E, F, T, S), T_1 < T < T_2. \tag{1}$$

$$\mathit{holdsAt}(F, T_2, S) \leftarrow \mathit{happens}(E, T_1, S), \mathit{initiates}(E, F, T_1, S), T_1 < T_2, \mathit{not\ clipped}(T_1, F, T_2, S). \tag{2}$$

$$\mathit{holdsAt}(F, T, S) \leftarrow \mathit{initially}(F, S), \mathit{not\ clipped}(0, F, T, S). \tag{3}$$

$$\mathit{happens}(E, T, S) \leftarrow \mathit{attempt}(E, T, S), \mathit{not\ impossible}(E, T, S). \tag{4}$$

$$\mathit{happens}(E, T, S) \leftarrow \mathit{attempt}(E, T, S), \mathit{triggered}(E, T, S). \tag{5}$$

$$\mathit{impossible}(E2, T, S) \leftarrow \mathit{triggered}(E1, T, S),\ \ E1 \neq E2. \tag{6}$$

$$\leftarrow \mathit{impossible}(E, T, S), \mathit{triggered}(E, T, S). \tag{7}$$

The three axioms (1)–(3) describe general principles for deciding when fluents hold or do not hold at particular time-points.[2] They formalise the commonsense law of inertia which states that a fluent that is true remains to hold until a terminating event occurs and vice versa. The two axioms (4) and (5) capture the semantics of event pre-conditions and event trigger-conditions respectively. The rule (4) states that an event E *cannot* happen if its pre-conditions are not satisfied (i.e. *impossible* is true). The rule (5), on the other hand, declares that an event E *must* happen if its trigger-conditions have been satisfied. The rule (6) indicates that if an event e is triggered at time point t in scenario s then all other events must be impossible at that time point in that scenario. The last rule (7) captures the semantic relationship between trigger-conditions and pre-conditions.[3] It states if an event e is impossible at time point t in scenario s, then it cannot be triggered at t in s.

3. The approach

In this section we show how ILP can be used to extend a partial system specification using information from a given set of scenarios. The learning problem is formalised in terms of an LTL specification. The given specification and scenarios are transformed, by means of a sound translation, into a non-monotonic ILP problem using an EC formalisation which is then used to find a set of requirements that together with the partial specification account for the example scenarios.

3.1. Problem Description

Our aim is to develop an approach for extending an incomplete system specification with two types of operational requirements, namely event *pre-conditions* and *trigger-conditions* using information provided by user-defined scenarios. Scenarios are sequences of events with a positive or negative label, according to whether it is possible or impossible for the last event in the sequence to occur after the previous events have occurred. Initial knowledge of the system is given by a partial system specification expressed in a restricted LTL syntax. Since LTL is used to represent system behaviours and their effects on the system and its environment, it is assumed that each fluent $f \in P_f$ is associated with two disjoint sets I_f and T_f called the f-*Initiating* set and f-*Terminating* set of event propositions respectively. Moreover, only a special class of LTL models are considered in which the labelled transition system T has a single initial state and the valuation function V is defined to capture the dependencies between fluent propositions and event propositions. For convenience, the notation E_{I_f} is used to represent the disjunction $\bigvee_{e \in I_f} e$ of f-Initiating events, and E_{T_f} for the disjunction $\bigvee_{e \in T_f} e$ of f-Terminating events. S_0 is instead used to represent the set of fluents $f \in P_f$ that are true in the initial system state s_0.

[2] Axioms (1)–(3) axioms are identical to those presented in [27] apart from the extra argument S for representing scenarios. Axioms (4)–(7) extend the formalism in [27] to support trigger-conditions and pre-conditions.

[3] These axioms can be made more realistic by restricting trigger-conditions and pre-conditions to system events, as opposed to environmental events over which the system has no control. However, for brevity, details are omitted.

A *system specification* describes the interactions between events relationship between events and fluents of the system and environment. As formalised in Definition 3.1 below, it contains *initial state axioms* (8)–(9), specifying the fluent propositions that are true (resp. false) in the initial state; *persistence axioms* (10)–(11), formalising the common-sense law of inertia that any fluent will remain true (resp. false) until a terminating (resp. initiating) event occurs that causes it to flip truth value; *effect axioms* (12)–(13), which describe the effect a set of f-Initiating and f-Terminating events has on a fluent f; and finally a set of *pre-condition axioms* (14), and a set of *trigger-condition axioms* (15), describing the conditions under which an event *cannot* occur and *must* occur, respectively.[4]

Definition 3.1. A *system specification* is an LTL theory consisting of

- a *positive initial state axiom* representing all fluent propositions that are true in S_0 (if any).

$$\bigwedge_{f_i \in S_0} f_i \tag{8}$$

- a *negative initial state axiom* representing all fluent propositions that are not true in s_0 (if any).

$$\bigwedge_{f_j \in (P_f - S_0)} \neg f_j \tag{9}$$

- a pair of *persistence axioms* for each fluent proposition $f \in P_f$.

$$\Box(f \rightarrow f \ W \ E_{T_f}) \tag{10}$$

$$\Box(\neg f \rightarrow \neg f \ W \ E_{I_f}) \tag{11}$$

- a pair of f-Initiating and f-Terminating *effect axioms* for each fluent proposition $f \in P_f$.

$$\Box(E_{I_f} \rightarrow f) \tag{12}$$

$$\Box(E_{T_f} \rightarrow \neg f) \tag{13}$$

- a set of *pre-condition axioms* (possibly empty).

$$\Box\Big(\bigwedge_{0 \leqslant i \leqslant n} (\neg) f_i \rightarrow \bigcirc \neg e\Big) \tag{14}$$

- a set of *trigger-condition axioms* (possibly empty).

$$\Box\Big(\bigwedge_{0 \leqslant i \leqslant m} (\neg) f_i \rightarrow \bigcirc e\Big). \tag{15}$$

A scenario is a finite sequence of events $\langle e_1, \ldots, e_n \rangle$ that describes a system's hypothetical behaviour from its initial state by specifying which events are initiated by the system in response to events occurring in the environment. A positive scenario, denoted $\langle e_1, \ldots, e_n \rangle^+$, allows the user to state that there should be at least one path in a model of the system specification where $\langle e_1, \ldots, e_n \rangle$ is accepted as an input. A negative scenario, denoted $\langle e_1, \ldots, e_n \rangle^-$, allows the user to state that there should be no paths in any model of the system specification where $\langle e_1, \ldots, e_n \rangle$ is accepted as an input. These notions are formalised by associating each scenario with an LTL formula, called a scenario property, as defined below.

Definition 3.2. A *scenario* is

- either a *positive scenario* of the form

$$\langle e_1, \ldots, e_m \rangle^+ \text{ representing the property } \bigwedge_{1 \leqslant i \leqslant m-1} \bigcirc^i e_i \wedge \bigcirc^m e_m \tag{16}$$

- or a *negative scenario* of the form

$$\langle e_1, \ldots, e_n \rangle^- \text{ representing the property } \bigwedge_{1 \leqslant i \leqslant n-1} \bigcirc^i e_i \rightarrow \bigcirc^n \neg e_n. \tag{17}$$

[4] Note that the notion of pre-condition used in this paper differs from the standard terminology, in the sense that the negation of the antecedent of our pre-condition axioms corresponds to the pre-condition for the event occurring.

Because the property associated with a positive scenario is expected to hold in at least one path of a model, it will be called an *existential scenario property* and denoted as sp_{ex}. Since the property associated with a negative scenario is expected to hold in all paths of a model, it will be called a *universal scenario property*, and denoted as sp_u. For convenience, scenarios will often be identified with their associated property.

Given the above formalisation, we can now define the task of learning pre-conditions and trigger-conditions.

Definition 3.3. Let *Spec* be a system specification, $M = \langle T, V\rangle$ be a model of *Spec*, *Scen* be a set of universal scenario properties $\{sp_{u_i}\}$ and existential scenario properties $\{sp_{ex_j}\}$. A set $Pre \cup Trig$ of pre-condition and trigger-condition axioms is a *correct extension* of *Spec* with respect to *Scen* iff

- $Spec \cup Pre \cup Trig \models_M sp_u$ for each $sp_{u_i} \in Scen$
- $Spec \cup Pre \cup Trig \not\models_M \neg sp_{ex}$ for each $sp_{ex_j} \in Scen$.

Intuitively, finding a correct extension of a given specification with respect to given existential and universal scenario properties means *refining* the models of the original *Spec* by removing unwanted or undesirable traces. Given a model $M = (T, V)$ of a specification *Spec*, adding a pre-condition axiom to *Spec* has the effect of removing from T all those (sub-)paths that satisfy the prefix of the pre-condition and have as subsequent transition one labelled with the event in the consequence of the pre-condition axiom. On the other hand, adding a trigger-condition axiom to the *Spec* means removing from T all those (sub-)paths that satisfy the prefix of the trigger-condition and have as subsequent transitions those labelled with an event different from the event in the consequence of trigger-condition axiom.

3.2. Translating LTL specifications into EC logic programs

In order to apply ILP to the task of learning correct extensions, the LTL system specification and scenario properties of the form defined above are translated into EC normal logic program, where the sorts of event and fluent are given by the set P_e of event propositions and the set P_f of fluent proposition of the LTL language; time points correspond to the positions in the paths of a model M, and a scenario constant is introduced for every universal and existential scenario property in *Scen*. The translation of a system specification *Spec* into such a correspond EC program Π is formally defined below.

Definition 3.4. Given a system specification *Spec* expressed in LTL, the *corresponding* logic program $\Pi = \tau(Spec)$ is defined as the program containing the following clauses:

- the fact $initially(f_i, S)$ for each fluent f_i appearing in a positive initial state axiom of the form $\bigwedge_{f_i \in S_0} f_i$
- the fact $initiates(e_i, f, T, S)$ for each f-Initiating event $e_i \in I_f$ appearing in an f-Initiating effect axiom of the form $\Box(E_{I_f} \rightarrow f)$
- the fact $terminates(e_i, f, T, S)$ for each f-Terminating event $e_i \in T_f$ appearing in a f-Terminating effect axiom of the form $\Box(E_{T_f} \rightarrow \neg f)$
- the clause $impossible(e, T, S) \leftarrow \bigwedge_{0\leqslant i\leqslant k}(not)holdsAt(f_i, T, S)$ for each pre-condi-tion axiom of the form

$$\Box\left(\bigwedge_{0\leqslant i\leqslant k} (\neg) f_i \rightarrow \bigcirc\neg e\right)$$

- the clause $triggered(e, T, S) \leftarrow \bigwedge_{0\leqslant i\leqslant l}(not)holdsAt(f_i, T, S)$ for each trigger-condition axiom of the form

$$\Box\left(\bigwedge_{0\leqslant i\leqslant l} (\neg) f_i \rightarrow \bigcirc e\right)$$

- the EC core axioms (1)–(7).

Note that negative initial state axioms (9) and persistence axioms (10) and (11) of an LTL specification are all implicitly captured by the stable model interpretation of the EC core axioms. Moreover, the semantics of the temporal operator $\Box$ is captured by the implicit universal quantification over the time variable T that appears in the *initiates* and *terminates* facts and in the *impossible* and *triggered* rules. Theorem 3.1 below states that the translation τ is sound, i.e. for any path σ in a given model M of *Spec* there is a corresponding narrative *Nar* such that the program $\Pi = \tau(Spec) \cup Nar$ satisfies the same fluent and event formulae as in σ.

Theorem 3.1. *Given an LTL language with propositions $P = P_e \cup P_f$, let Spec be a system specification in the given language and $M = \langle T, V\rangle$ a model of Spec. Let $\sigma = s_0\mathcal{R}_{e_1}s_1, \ldots, s_{n-1}\mathcal{R}_{e_n}s_n$ be a path in T and let Nar be the set of facts of the form $attempt(e_i, i-1, \sigma)$ for each event e_i in σ. Let Π be the EC logic program $\Pi = \tau(Spec) \cup Nar$ with a unique stable model I. Then, for any position $j \geqslant 0$ we have that $\sigma, j \models e$ iff $happens(e, j-1, \sigma) \in I$, for any event $e \in P_e$, and $\sigma, j \models f$ iff $holdsAt(f, j, \sigma) \in I$, for any fluent $f \in P_f$.*

Proof. The proof is by structural induction on position j in σ. We consider two base cases for $j = 0$ and $j = 1$. The first base case is to cover the satisfiability of fluent propositions in the initial state (i.e. at position 0) and the second is to cover the satisfiability of event propositions which start from position 1. Let Π^I be the reduct of program Π, so that I is its minimal (Herbrand) model.

Base cases

(1) $j = 0$:
The case $\sigma, 0 \models e$ iff $\mathit{happens}(e, -1, \sigma) \in I$ is trivially true. By Definition 2.6 of satisfiability, $\sigma, 0 \not\models e$ for any $e \in P_e$; also for any $e \in P_e$, no ground atom $\mathit{happens}(e, -1, \sigma)$ is in I, since the time constant -1 is outside the scope of the EC time sort T.
We show that $\sigma, 0 \models f$ iff $\mathit{holdsAt}(f, 0, \sigma) \in I$. We consider the "if case" first. $\sigma, 0 \models f$ implies $f \in S_0$.[5] Hence the program Π, and therefore I, contains the ground atom $\mathit{initially}(f, \sigma)$. Moreover, $\mathit{clipped}(0, f, 0, \sigma) \notin I$ since there is no t, $0 < t < 0$. Then Π^I contains the ground clause $\mathit{holdsAt}(f, 0, \sigma) \leftarrow \mathit{initially}(f, \sigma)$ and therefore $\mathit{holdsAt}(f, 0, \sigma) \in I$. We now consider the "only if" case. Reasoning by contradiction, we assume that $\sigma, 0 \not\models f$. Hence, f does not appear in the positive initial state axiom of *Spec*, and therefore Π does not contain the ground atom $\mathit{initially}(f, \sigma)$. Moreover, since no $\mathit{happens}(e, -1, \sigma)$ is in Π, for any $e \in P_e$, then Π^I does not contain ground definitions of $\mathit{holdsAt}(f, 0, \sigma)$. Hence $\mathit{holdsAt}(f, 0, \sigma) \notin I$, giving a contradiction.

(2) $j = 1$:
We show that $\sigma, 1 \models e$ iff $\mathit{happens}(e, 0, \sigma) \in I$ for any event e. We consider the "if case" first, and assume that $\sigma, 1 \models e$, for an arbitrary event e. Then there is a transition $s_0 \mathcal{R}_e s_1$ in σ. This means that the antecedents of any precondition in *Spec* of the form (14) for the event e are false at position 0, and the antecedents of any trigger-condition in *Spec* of the form (15) for the event e are true at position 0. By the previous base case, the same holds for the corresponding set of EC pre-condition and trigger-condition clauses, respective. Therefore, $\mathit{impossible}(e, 0, \sigma) \notin I$ and $\mathit{triggered}(e, 0, \sigma) \in I$. Hence, given the rules (4) and (5) in Π^I and the fact that $\mathit{attempt}(e, 0, \sigma) \in I$, by assumption, we can conclude that $\mathit{happens}(e, 0, \sigma) \in I$. We consider now the "only if" case. Assume that $\mathit{happens}(e, 0, \sigma) \in I$ and that, reasoning by contradiction, $\sigma, 1 \not\models e$. Since e is not satisfied at position 1 of σ, Π does not contain the fact $\mathit{attempt}(e, 0, \sigma)$. From $\mathit{attempt}(e, 0, \sigma) \notin I$ it follows that $\mathit{happens}(e, 0, \sigma) \notin I$ which leads to a contradiction.
We show that $\sigma, 1 \models f$ iff $\mathit{holdsAt}(f, 1, \sigma) \in I$. We assume that $\sigma, 1 \models f$ for an arbitrary fluent f. This implies that f is either (a) initially true (i.e $\sigma, 0 \models f$) and no f-Terminating event occurs at 1 or (b) has been initiated by an f-Initiating event at 1. Consider the case (a). f true in the initial state implies that Π^I contains the atom $\mathit{initially}(f, \sigma)$. Moreover, since $\sigma, 1 \not\models e_{T_f}$ for any terminating event e_{T_f} then, by the first base case, $\mathit{happens}(e_{T_f}, 0, \sigma) \notin I$ and thus $\mathit{clipped}(0, f, 1, \sigma) \notin I$. Π^I contains in this case the ground clause $\mathit{holdsAt}(f, 1, \sigma) \leftarrow \mathit{initially}(f, \sigma)$. Hence, $\mathit{holdsAt}(f, 1, \sigma) \in I$.
We now consider the (b) case. We have that $\sigma, 1 \models e_{I_f}$, therefore $\mathit{happens}(e_{I_f}, 0, \sigma) \in I$. Given that only one event can be true at any single position in a path, we know that $\sigma, 1 \not\models e'$ for any $e' \in P_e - \{e_{I_f}\}$ including any f-terminating event. Hence, $\mathit{happens}(e_{T_f}, 0, \sigma) \notin I$ for any terminating event e_{T_f}. This implies that $\mathit{clipped}(0, f, 1, \sigma) \notin I$ and $\mathit{holdsAt}(f, 1, \sigma) \leftarrow \mathit{initiates}(e_{I_f}, f, 0, \sigma), \mathit{happens}(e_{I_f}, 0, \sigma), 0 < 1$ is in Π^I. Since $\mathit{initiates}(e_{I_f}, f, 0, \sigma) \in I$ also $\mathit{holdsAt}(f, 1, \sigma) \in I$. The proof of the "only if" case is similar.

Induction Hypothesis (IH):
We assume that for any position k, $0 \leqslant k \leqslant j - 1$ and for any event e and fluent f, we have $\sigma, k \models e$ iff $\mathit{happens}(e, k - 1, \sigma) \in I$, and $\sigma, k \models f$ iff $\mathit{holdsAt}(f, k, \sigma) \in I$. We want to show this is true for position $k = j$.

We want to show that $\sigma, j \models e$ iff $\mathit{happens}(e, j - 1, \sigma) \in I$. We consider the "if case" first and assume that $\sigma, j \models e$. This means that a transition $s_{j-1} \mathcal{R}_e s_j$ exist in σ. Then the antecedent of any precondition in *Spec* of the form (14) for the event e is false at position $j - 1$, and the antecedents of any trigger-condition in *Spec* of the form (15) for the event e is true at position $j - 1$. Given that the program Π contains pre-condition and trigger-condition clauses on e, by (IH) their respective antecedents are also satisfied at time point $j - 1$ in σ. Then $\mathit{impossible}(e, j - 1, \sigma) \notin I$ for all pre-condition clauses on e and $\mathit{triggered}(e, j - 1, \sigma) \in I$. Now given that $\mathit{attempt}(e, j - 1, \sigma) \in I$, we can conclude that $\mathit{happens}(e, j - 1, \sigma) \in I$. We now need to show the "only if" case. We assume that $\mathit{happens}(e, j - 1, \sigma) \in I$ and, reasoning by contradiction, that $\sigma, j \not\models e$. Since e is not satisfied at position j of σ, Π does not contain the fact $\mathit{attempt}(e, j - 1, \sigma)$. Therefore, $\mathit{happens}(e, j - 1, \sigma) \notin I$ which is a contradiction.

We need to prove that $\sigma, j \models f$ iff $\mathit{holdsAt}(f, j, \sigma) \in I$. We consider the "if case" first and assume that $\sigma, j \models f$. Then either (a) $\sigma, 0 \models f$ and $\sigma, k \not\models e_{T_f}$ for all $1 \leqslant k \leqslant j$ and f-Terminating events e_{T_f}, or (b) $\sigma, k \models e_{I_f}$, for some k such that $1 \leqslant k \leqslant j$ and for all l, with $k < l \leqslant j$, $\sigma, l \not\models e_{T_f}$ for all f-terminating events. If (a) is the case, then we know that $\mathit{initially}(f, \sigma) \in I$ and for all $0 \leqslant k \leqslant j$, $\sigma, k \not\models e_{T_f}$. Then by (IH) $\mathit{happens}(e_{T_f}, k - 1, \sigma) \notin I$, for all $0 \leqslant k \leqslant j$, and therefore $\mathit{clipped}(0, f, k, \sigma) \notin I$. The transformed program Π^I would therefore include $\mathit{holdsAt}(f, j, \sigma) \leftarrow \mathit{initially}(f, \sigma)$ and hence $\mathit{holdsAt}(f, j, \sigma) \in I$.

[5] Our *Spec* is assumed to be complete on its initial state.

We consider case (b). Given that an f-Initiating event has occurred at some k, with $1 \leqslant k \leqslant j$ (i.e. $\sigma, k \models e_{I_f}$) then, by (IH), $happens(e_{I_f}, k-1, \sigma) \in I$. Moreover, since no f-Terminating event occurs at any l with $k < l \leqslant j$, Π^I does not contain any atom of the form $attempt(e_{T_f}, l-1, \sigma)$ and therefore $happens(e_{T_f}, l-1, \sigma) \notin I$. This means that the ground atom $clipped(k-1, f, j, \sigma) \notin I$. Thus, Π^I contains the ground rule

$$holdsAt(f, j, \sigma) \leftarrow initiates(e_{I_f}, f, k-1, \sigma), happens(e_{I_f}, k-1, \sigma), k-1 < j.$$

Hence, $holdsAt(f, j, \sigma) \in I$.

We consider now the "only if" case and assume that $holdsAt(f, j, \sigma) \in I$. We want to show that $\sigma, j \models f$. Reasoning by contradiction, suppose that $\sigma, j \not\models f$. Hence either (a) f is initially false and has not been initiated at any position k, with $0 \leqslant k \leqslant j$, or (b) for any position k where f is true, with $0 \leqslant k \leqslant j-1$, there is a later position l, with $k < l \leqslant j$ such that $\sigma, l \models e_{T_f}$. We first consider the case (a). Since $\sigma, 0 \not\models f$, $initially(f, \sigma) \notin I$. Furthermore, because $\sigma, k \not\models e_{I_f}$ for any f-Initiating event in P_e $happens(e_{I_f}, k-1, \sigma) \notin I$. Thus, under the stable model semantics $holdsAt(f, j, \sigma) \notin I$ which is a contradiction.

Consider now the (b) case and let k be the last position in σ such that $\sigma, k \models f$ where $0 < k < j-1$ and such that there is a later position l where $k < l \leqslant j$ where $\sigma, l \models e_{T_f}$ for some f-Terminating event e_{T_f}. Then by (IH) $happens(e_{T_f}, l-1, \sigma) \in I$ and hence $clipped(k-1, f, j, \sigma) \in I$. Thus under the stable model semantics the following ground rules are omitted in Π^I:

$$holdsAt(f, j, \sigma) \leftarrow initially(f, \sigma), not\ clipped(0, f, j, \sigma)$$

$$holdsAt(f, j, \sigma) \leftarrow initiates(e_{I_f}, f, k-1, \sigma), happens(e_{I_f}, k-1, \sigma), not\ clipped(k-1, f, j, \sigma)$$

Hence, $holdsAt(f, j, \sigma) \notin I$ which is a contradiction. □

The following corollary states the soundness of the translation with respect to a set of paths.

Corollary 3.1. *Let Spec be a system specification and let $M = \langle T, V \rangle$ be a model of Spec. Let $\Sigma = \{\sigma_h \mid 1 \leqslant h \leqslant m\}$ be a set of paths in T. Let Nars be the set of narratives obtained from each σ_h of the form attempt$(e_i, i-1, \sigma_h)$ for each event e_i in σ_h. Let $\Pi = \tau(Spec) \cup Nars$ be the EC logic program with (unique) stable model I. Then for each σ_h and for any event $e \in P_e$ and position $j \geqslant 0$, we have $\sigma_h, j \models e$ iff happens$(e, j-1, \sigma_h) \in I$; and for any fluent $f \in P_f$, we have $\sigma_h, j \models f$ iff holdsAt$(f, j, \sigma_h) \in I$.*

3.3. Learning requirements using ILP

ILP is concerned with the task of learning a hypothesis H that explains a set of examples E with respect to a background theory B. In the context of learning requirements, we are given a partial specification B and a set of scenarios E, and the task is to learn a set H of operational requirements such that $B \cup H \models E$ (under the stable model semantics). In the EC formalism, event pre-conditions are represented as clauses with *impossible* in the head and *holdsAt* literals in the body

$$impossible(e, T, S) \leftarrow (not)\ holdsAt(f_1, T, S), \ldots, (not) holdsAt(f_n, T, S) \tag{18}$$

while event trigger-conditions are represented as clauses of the form

$$triggered(e, T, S) \leftarrow (not)\ holdsAt(f_1, T, S), \ldots, (not)\ holdsAt(f_m, T, S). \tag{19}$$

Hence, the task of learning requirements is the process of generating hypotheses of the form above from a partial specification and a set of scenario properties which, respectively, comprise the background and examples. The function τ can be used to translate an initial LTL specification into an ILP theory. To fully define the inductive learning task a corresponding translation from scenario properties to ILP example must be specified. As shown in the definition below, scenario properties contribute to the background theory as well as to the examples. The translation depends on the event for which the pre-condition (resp. trigger-condition) axiom is to be learnt. In what follows, it is assumed that pre-conditions (resp. trigger-conditions) axioms are to be learnt for the last event of each universal scenario and existential scenario property. Therefore, each universal scenario property produces a sequence of facts stating that certain events do happen followed by the fact that some particular event should not (resp. another event must) happen immediately afterward, and each existential scenario property simply states that a certain sequence of events does happen.

Definition 3.5. Given a system specification *Spec* and a set *Scen* consisting of universal and existential scenario properties, the *EC translation* $\tau(Spec, Scen)$ is the pair (B, E) of EC programs constructed as follows:

- For each universal scenario property $sp_u = (\bigwedge_{1 \leqslant i \leqslant n-1} \bigcirc^i e_i \rightarrow \bigcirc^n \neg e_n)$ in *Scen*.
  - . E includes $n-1$ facts of the form $happens(e_i, i-1, sp_u)$ with $1 \leqslant i \leqslant n-1$.
  - . E includes 1 fact of the form $not\ happens(e_n, n-1, sp_u)$.
  - . B includes n facts $attempt(e_i, i-1, sp_u)$ with $1 \leqslant i \leqslant n$.

- For each existential scenario property $sp_{ex} = \bigwedge_{1 \leqslant i \leqslant k} \bigcirc^i e_i$ in *Scen*.
  . E includes k facts of the form $happens(e_i, i-1, sp_{ex})$ with $1 \leqslant i \leqslant k$.
  . B includes k facts of the form $attempt(e_i, i-1, sp_{ex})$ with $1 \leqslant i \leqslant k$.
- B includes all facts and rules in $\tau(Spec)$.

As shown in the definition above, given a partial specification *Spec* and a set of scenario properties *Scen*, the translation τ gives a corresponding EC program consisting of a background theory B and a set of examples E where $(B, E) = \tau(Spec, Scen)$.

The utility of the transformation is demonstrated by Theorem 3.2, which shows that τ can be used to compute correct extensions via non-monotonic ILP. In particular, given a partial specification *Spec*, set of scenario properties *Scen*, the corresponding EC programs $(B, E) = \tau(Spec, Scen)$, the hypothesis space HS is defined as the set of clauses of the form (18) and (19). Theorem 3.2 states that, given a set *Scen* of universal and existential scenario properties, any non-monotonic ILP solution to this problem can be translated back into a correct LTL extension of the initial *Spec* with respect *Scen*.

Theorem 3.2. *Let Spec be a system specification, let Scen be a set of universal and existential scenario properties such that $Spec \not\models sp_{u_i}$ for all $sp_{u_i} \in Scen$, and $Spec \not\models \neg sp_{ex_j}$ for all $sp_{ex_j} \in Scen$. Let $(B, E) = \tau(Spec, Scen)$ be the corresponding EC program, and let HS be the set of clauses of the form (18) and (19)—i.e. the set of all EC pre-condition and trigger-condition rules. Then, for any inductive generalisation H of E w.r.t. B and HS, the corresponding set $Pre \cup Trig = \tau^{-1}(H)$ of LTL pre-condition and trigger-condition axioms is a correct extension of Spec with respect to Scen.*

Proof. Given $Pre \cup Trig = \tau^{-1}(H)$, we want to show that $Spec \cup Pre \cup Trig \models sp_{u_i}$ and $Spec \cup Pre \cup Trig \not\models \neg sp_{ex_j}$ for each sp_{u_i} and sp_{ex_j} in *Scen*.

Consider the second case first.

Suppose that $Spec \cup Pre \cup Trig \models \neg sp_{ex_j}$, for some existential scenario property sp_{ex_j} of the form $\bigwedge_{1 \leqslant k \leqslant m_j - 1} \bigcirc^k e_k \wedge \bigcirc^{m_j} e$. This means that $Spec \cup Pre \cup Trig$ does not contain a path of the form $\sigma_{ex_j} = s_0 \mathcal{R}_{e_1} s_1, \ldots, s_{m_j-1} \mathcal{R}_e s_{m_j}$. So $Spec \cup Pre \cup Trig$ includes a precondition for an event e_h, with $1 \leqslant h \leqslant m_j - 1$, or for e.

Assume $Spec \cup Pre \cup Trig$ includes a precondition for an event e_h. Consider *Spec* to include such a pre-condition. Then the path $\sigma_{ex} = s_0 \mathcal{R}_{e_1} s_1, \ldots, s_{h-1} \mathcal{R}_{e_h} s_h$ would also not exist already in the model M, which is inconsistent with our given initial assumption. Therefore, $Pre \cup Trig$ includes a precondition on e_h where the antecedent is satisfied at state s_{h-1} of σ_{ex} and therefore $\sigma_{ex} \models \bigcirc^h \neg e_h$. Now, the program $\Pi = (B \cup H)$ is equal to $\tau(Spec \cup Pre \cup Trig) \cup Nars$ with respect to the path(s) associated with the properties in *Scen*. So by Corollary 3.1 we have that $B \cup H \models happens(e_k, k-1, sp_{ex_j})$ where $1 \leqslant k \leqslant h-1$, $B \cup H \models happens(e_k, k-1, sp_{ex_j})$ where $h+1 \leqslant k \leqslant m_j - 1$, $B \cup H \models happens(e, m_j - 1, sp_{ex_j})$ and $B \cup H \models$ *not* $happens(e_h, h-1, sp_{ex_j})$, which is in contradiction with the fact that $B \cup H \models E$.

Then $Spec \cup Pre \cup Trig$ includes a precondition for e. Consider *Spec* to include such a pre-condition. Therefore the path $s_0 \mathcal{R}_{e_1} s_1, \ldots, s_{m_j-1} \mathcal{R}_e s_{m_j}$ would also not exist already in the model M, which is inconsistent with our given initial assumption. Moreover, $Pre \cup Trig$ includes a precondition on e. Consider the sub-path $s_0 \mathcal{R}_{e_1} s_1, \ldots, s_{m_j-2} \mathcal{R}_e s_{m_j-1}$ in which e does not occur. Now, the program $\Pi = (B \cup H)$ is equal to $\tau(Spec \cup Pre \cup Trig) \cup Nars$ with respect to the path(s) associated with the properties in *Scen*. So by Corollary 3.1 we have that $B \cup H \models happens(e_k, k-1, sp_{ex_j})$ where $1 \leqslant k \leqslant m_j - 1$ and $B \cup H \models$ *not* $happens(e, m_j - 1, sp_{ex_j})$, which is in contradiction with the fact that $B \cup H \models E$.

We now show that $Spec \cup Pre \cup Trig \models sp_{u_i}$ for all sp_{u_i} in *Scen*.

Reasoning by contradiction, we assume that $Spec \cup Pre \not\models sp_{u_i}$ for some sp_{u_i} of the form $\bigwedge_{1 \leqslant l \leqslant n_i - 1} \bigcirc^l e_l \rightarrow \bigcirc^{n_i} \neg e$. This means that the model of $Spec \cup Pre \cup Trig$ includes the path $s_0 \mathcal{R}_{e_1} s_1, \ldots, s_{n_i-1} \mathcal{R}_e s_{n_i}$. Now, the program $\Pi = (B \cup H)$ is equal to $\tau(Spec \cup Pre \cup Trig) \cup Nars$ with respect to all path(s) associated to the scenario properties in *Scen*. By Corollary 3.1 we have that $B \cup H \models happens(e_l, l-1, sp_{u_i})$ where $1 \leqslant l \leqslant n_i - 1$ and $B \cup H \models happens(e, n_i - 1, sp_{u_i})$, which is in contradiction with the fact that $B \cup H \models E$. $\square$

In other words, for a given B and E, any inductive solution of the form (18) and (19) once translated back into LTL and added to the original partial specification will have the effect of eliminating all paths in T which violate all sp_{u_i} while maintaining at least one path satisfying the corresponding sp_{ex_j} properties.

4. Case study: A Mine Pump control system

This section presents an extension of the case study used in [1] as an application of the learning approach proposed in this paper to a event-driven system involving a Mine Pump Controller [8]. This is a system that is supposed to monitor and control water levels in a mine, to prevent water overflow. It is composed of a pump for pumping mine-water up to the surface and sensors for monitoring the water levels and methane percentage. The pump must be activated once the water has reached pre-set high water level and deactivated once it reaches low water level. Moreover, the pump must be switched off if the percentage of methane in the mine exceeds a certain critical limit.

An initial partial system specification *Spec* is given along with a set of universal and existential scenario properties, written in an LTL language with fluent propositions $P_f = \{pumpOn, criticalMethane, highWater\}$ and event propositions

$P_e = \{turnPumpOn, turnPumpOff, signalAlarm, signalNotAlarm, signalCriticalMethane, signalNotCriticalMethane, signalHighWater, signalLowWater\}$. The specification includes information about the initial state of the system, persistence axioms, effect axioms and a single trigger-condition axiom, all formalised as follows:

$$(\neg criticalMethane \wedge \neg pumpOn \wedge \neg highWater) \tag{20}$$

$$\Box(criticalMethane \rightarrow (criticalMethane\ \ W\ \ signalNotCriticalMethane)) \tag{21}$$

$$\Box(\neg criticalMethane \rightarrow (\neg criticalMethane\ \ W\ \ signalCriticalMethane)) \tag{22}$$

$$\Box(pumpOn \rightarrow (pumpOn\ \ W\ \ turnPumpOff)) \tag{23}$$

$$\Box(\neg pumpOn \rightarrow (\neg pumpOn\ \ W\ \ turnPumpOn)) \tag{24}$$

$$\Box(highWater \rightarrow (highWater\ \ W\ \ signalLowWater)) \tag{25}$$

$$\Box(\neg highWater \rightarrow (\neg highWater\ \ W\ \ signalHighWater)) \tag{26}$$

$$\Box(signalCriticalMethane \rightarrow criticalMethane) \tag{27}$$

$$\Box(signalNotCriticalMethane \rightarrow \neg criticalMethane) \tag{28}$$

$$\Box(signalHighWater \rightarrow highWater) \tag{29}$$

$$\Box(signalLowWater \rightarrow \neg highWater) \tag{30}$$

$$\Box(turnPumpOn \rightarrow pumpOn) \tag{31}$$

$$\Box(turnPumpOff \rightarrow \neg pumpOn) \tag{32}$$

$$\Box(criticalMethane \wedge pumpOn \rightarrow \bigcirc turnPumpOff). \tag{33}$$

Eq. (20) defines the negative initial state of the system, Eqs. (21)–(26) specify the persistence axioms and Eqs. (27)–(32) define the effect axioms and finally Eq. (33) specifies a trigger-condition axiom. Note that because all fluents are assumed to be initially false, the specification does not include a positive initial state axiom.

A consistent set of positive and negative scenarios for this system are given by the following formulae:

$$sp_{u_1} = \langle signalCriticalMethane, signalHighWater, turnPumpOn\rangle^- \tag{34}$$

$$sp_{u_2} = \langle signalCriticalMethane, signalAlarm\rangle^- \tag{35}$$

$$sp_{u_3} = \langle signalLowWater, turnPumpOn\rangle^- \tag{36}$$

$$sp_{ex_1} = \langle signalNotCriticalMethane, signalHighWater, turnPumpOn\rangle^+. \tag{37}$$

Applying the translation τ to the specification and scenario properties above results in an ILP theory B composed of the EC core axioms and the following clauses:

$initiates(signalCriticalMethane, criticalMethane, T, S).$
$terminates(signalNotCriticalMethane, criticalMethane, T, S).$
$initiates(signalHighWater, highWater, T, S).$
$terminates(signalLowWater, highWater, T, S).$
$initiates(turnPumpOn, pumpOn, T, S).$
$terminates(turnPumpOff, pumpOn, T, S).$
$triggered(turnPumpOff, T, S) \leftarrow holdsAt(criticalMethane, T, S), holdsAt(pumpOn, T, S).$

$attempt(signalCriticalMethane, 0, sp_{u_1}).$ $attempt(signalHighWater, 1, sp_{u_1}).$
$attempt(turnPumpOn, 2, sp_{u_1}).$ $attempt(signalCriticalMethane, 0, sp_{u_2}).$
$attempt(signalAlarm, 1, sp_{u_2}).$ $attempt(signalLowWater, 0, sp_{u_3}).$
$attempt(turnPumpOn, 1, sp_{u_3}).$ $attempt(signalNotCriticalMethane, 0, sp_{ex_1}).$
$attempt(signalHighWater, 1, sp_{ex_1}).$ $attempt(turnPumpOn, 2, sp_{ex_1}).$

In addition, the translation produces the following set of ILP examples (for convenience, only the literals containing system events are shown here as the environmental events are trivially explained by the prior theory):

$not\ happens(turnPumpOn, 2, sp_{u_1}).$ $not\ happens(signalAlarm, 1, sp_{u_2}).$
$not\ happens(turnPumpOn, 1, sp_{u_3}).$ $happens(turnPumpOn, 2, sp_{ex_1}).$

In order for XHAIL to learn pre- and trigger-conditions, the following mode declarations are used.

$$\begin{aligned}&\mathit{modeh}(*, \mathit{impossible}(\#\mathit{event}, +\mathit{time}, +\mathit{scenario}))\\&\mathit{modeh}(*, \mathit{triggered}(\#\mathit{event}, +\mathit{time}, +\mathit{scenario}))\\&\mathit{modeb}(*, \mathit{holdsAt}(\#\mathit{fluent}, +\mathit{time}, +\mathit{scenario}))\\&\mathit{modeb}(*, \mathit{not}\ \mathit{holdsAt}(\#\mathit{fluent}, +\mathit{time}, +\mathit{scenario})).\end{aligned} \tag{38}$$

These mode declarations ensure the hypothesis space *HS* consists of clauses of the form (18) and (19) with atoms $\mathit{impossible}(e, T, S)$ and $\mathit{triggered}(e, T, S)$ in the head and literals of the form $(\mathit{not})\ \mathit{holdsAt}(e, T, S)$ in the body.

When applied to the inputs B, E and M, XHAIL computes the smallest size of abductive explanation and returns 27 *minimal* explanations each containing 3 atoms. One of these is the set $\Delta =$

$$\begin{aligned}&\mathit{triggered}(\mathit{turnPumpOff}, 2, sp_{u_1})\\&\mathit{triggered}(\mathit{signalAlarm}, 1, sp_{u_2})\\&\mathit{triggered}(\mathit{turnPumpOff}, 1, sp_{u_2})\\&\mathit{impossible}(\mathit{turnPumpOn}, 1, sp_{u_3}).\end{aligned} \tag{39}$$

The abductive computation of the predicate *triggered* makes use of the rule (6) of the EC core axioms. This enables the explanation of $\mathit{not}\ \mathit{happen}(\mathit{turnPumpOn}, 2, sp_{u_1})$ in E in terms of the abductive assumption $\mathit{triggered}(\mathit{turnPumpOff}, 2, sp_{u_1})$ for the event *turnPumpOff* different from the system event *turnPumpOn*. Similarly for the other triggered abductive assumptions.

The following set $K1$ is generated from the above abductive explanation.

$$\begin{aligned}&\mathit{triggered}(\mathit{turnPumpOff}, X1, X2) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2),\\&\qquad \mathit{not}\ \mathit{holdsAt}(\mathit{pumpOn}, X1, X2), \mathit{not}\ \mathit{holdsAt}(\mathit{highWater}, X1, X2).\\&\mathit{triggered}(\mathit{turnPumpOff}, X1, X2) \leftarrow \mathit{not}\ \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2),\\&\qquad \mathit{not}\ \mathit{holdsAt}(\mathit{highWater}, X1, X2), \mathit{not}\ \mathit{holdsAt}(\mathit{pumpOn}, X1, X2).\\&\mathit{impossible}(\mathit{turnPumpOn}, X1, X2) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2),\\&\qquad \mathit{holdsAt}(highWater, X1, X2), \mathit{not}\ \mathit{holdsAt}(\mathit{pumpOn}, X1, X2).\end{aligned} \tag{40}$$

Similarly, the following set $K2$ is computed as an alternative explanation.

$$\begin{aligned}&\mathit{triggered}(\mathit{turnPumpOff}, X1, X2) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2),\\&\qquad \mathit{not}\ \mathit{holdsAt}(\mathit{pumpOn}, X1, X2), \mathit{not}\ \mathit{holdsAt}(\mathit{highWater}, X1, X2).\\&\mathit{triggered}(\mathit{turnPumpOff}, X1, X2) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2),\\&\qquad \mathit{not}\ \mathit{holdsAt}(\mathit{pumpOn}, X1, X2), \mathit{not}\ \mathit{holdsAt}(\mathit{highWater}, X1, X2).\\&\mathit{triggered}(\mathit{signalAlarm}, X1, X2) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2),\\&\qquad \mathit{holdsAt}(\mathit{highWater}, X1, X2), \mathit{not}\ \mathit{holdsAt}(\mathit{pumpOn}, X1, X2).\end{aligned} \tag{41}$$

Both are generalised to give the following maximally compressive hypotheses $H1 =$

$$\begin{aligned}&\mathit{triggered}(\mathit{turnPumpOff}, X1, X2) \leftarrow \mathit{not}\ \mathit{holdsAt}(\mathit{highWater}, X1, X2).\\&\mathit{impossible}(\mathit{turnPumpOn}, X1, X2) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2).\end{aligned} \tag{42}$$

and $H2 =$

$$\begin{aligned}&\mathit{triggered}(\mathit{turnPumpOff}, X1, X2) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2),\\&\mathit{triggered}(\mathit{turnPumpOff}, X1, X2) \leftarrow \mathit{not}\ \mathit{holdsAt}(\mathit{highWater}, X1, X2).\\&\mathit{triggered}(\mathit{signalAlarm}, X1, X2) \leftarrow \mathit{holdsAt}(\mathit{criticalMethane}, X1, X2), \mathit{holdsAt}(\mathit{highWater}, X1, X2),\end{aligned} \tag{43}$$

corresponding to the correct extensions

$$\begin{aligned}&\Box(\neg \mathit{highWater} \to \bigcirc \mathit{turnPumpOff})\\&\Box(\mathit{criticalMethane} \to \bigcirc \neg \mathit{turnPumpOn})\end{aligned} \tag{44}$$

stating that the pump should not be turned on whenever the methane level is critical; and that it should be turned off whenever the water level is not above the threshold, and

$$\begin{aligned}&\Box(\mathit{criticalMethane} \to \bigcirc \mathit{turnPumpOff})\\&\Box(\neg \mathit{highWater} \to \bigcirc \mathit{turnPumpOff})\\&\Box(\mathit{criticalMethane} \wedge \mathit{highWater} \to \bigcirc \mathit{signalAlarm})\end{aligned} \tag{45}$$

stating that the pump should be turned off whenever the methane level is critical or whenever the water level is not high.

Any computed solution is guaranteed to be correct with respect to the given set of scenario properties. However, the choice among the possible sets of correct extensions is left to the engineer.

5. Related work

Other automated reasoning techniques have increasingly been used in requirements engineering [3,4,10,25]. Among these, the work most related to our approach is [10], where an *ad-hoc* inductive inference process is used in which high-level goals, expressed as temporal formulae, are derived from manually attuned scenarios provided by stakeholders. Each scenario is used to infer a set of goal assertions that explains it. Then each goal is added to the initial goal model, which is then analyzed using state-based analysis techniques (i.e. goal decomposition, conflict management and obstacle detection). The inductive inference procedure used in [10] is mainly based on pure generalisation of the given scenarios and does not take into account the given (partial) goal model. It is therefore a potentially unsound inference process by the fact that the generated goals may well be inconsistent with the given (partial) goal model. Our approach, on the other hand, allows the goals to be expressed as part of the background knowledge for the learning, thus constraining the system to produce only those requirements that are consistent with the given goals. Our learned required preconditions can therefore be automatically added to the existing goal model.

The work in [3] also proposes the use of inductive inference to generate behaviour models. It provides an automated technique for constructing LTSs from a set of user-defined scenarios. The synthesis procedure uses grammar induction to derive an LTS that covers all positive scenarios but none of the negative ones. The generated LTS can then be used for formal event-based analysis techniques (e.g. check against the goals expressed as safety properties). Our approach, on the other hand, uses the LTSA toolset [14] to generate the LTS models directly from (synchronous) goal models, so our LTS models are always guaranteed to satisfy the given goals.

The ILP task defined in this paper is somewhat related to some earlier work in [18] and [17], where the ILP systems Progol5 and Alecto were applied to the learning of domain-specific EC axioms. Like XHAIL, these procedures employ an abductive reasoning module to enable the learning of predicates distinct from those in the examples—an ability that is clearly required in this application. However, unlike XHAIL, they do not have a well-defined semantics for non-definite programs and their handling of negation is rather limited [24]. In fact, the inability of Progol5 and Alecto to reason abductively through nested negations means that neither of these systems can solve the case study presented in this paper. Some related approaches for inferring action theories from examples are presented in [13] and [22], which reduce learning in the Situation Calculus to a monotonic ILP framework. These approaches work by pre- and post-processing the inputs and outputs of a conventional Horn Clause ILP system. This technique is very efficient, but is not as general as our own approach. An alternative method for nonmonotonic ILP under the stable model semantics is proposed in [26], but cannot be used in our case study because it assumes the target predicate is the same as the examples. [26] also includes a thorough review of previous work on non-monotonic ILP.

6. Conclusion and future work

This paper presents a method for extending a partial system specification with event pre-conditions and trigger-conditions from information provided by user scenarios using ILP. This involves transforming the initial specification and scenarios from an LTL representation into an EC logic program which is then used by a non-monotonic ILP system to learn the missing requirements. By exploiting the semantic relationship between LTL and EC, the approach provides a sound ILP computational "back-end" to a temporal formalism familiar to Requirements Engineers.

The approach could be adapted to learning partial system descriptions from example. This would correspond to learning *initiates* and *terminates* EC rules. In this way we would be able to provide support for the computation of system specifications from scenarios, as well as operational requirements, allowing the stakeholders to convey such descriptions purely in terms of narrative-style scenarios of system behaviours, rather than LTL representations.

Furthermore, within the context of learning triggers, the above approach can be adapted to learn trigger-conditions directly from scenarios rather that the core axiom (6). In such cases, we would define other forms of universal and existential scenario properties. For instance, a universal scenario property for a trigger-condition would be $(\bigwedge_{1\leqslant i\leqslant n-1} \bigcirc^i e_i \rightarrow \bigcirc^n e)$ meaning that any sub-path that satisfies the prefix $\bigwedge_{1\leqslant i\leqslant n-1} \bigcirc^i e_i$ should immediately be followed by the event e. An existential universal scenario could be of the form $(\bigwedge_{1\leqslant i\leqslant m-1} \bigcirc^i e_i \wedge \bigcirc^m \neg e)$ meaning that there is at least one path that satisfies the prefix and in which e is not triggered immediately afterwards. In this case, the learning would compute a set of trigger-conditions that cover all universal scenario properties of the above form and is consistent with those captured by the existential one. Clearly, the translation function described in Definition 3.5 would need to be adjusted to capture the different semantics in the EC program between the translation of universal scenario property described above and the positive existential scenario property of the form (16). Similarly, it would need to reflect the difference between the existential scenario property described above and the negative universal scenario property of the form (17). Furthermore, additional core axioms may be required to distinguish between events occurring due to a trigger-condition being satisfied (i.e. captured by the EC rule (5)) and ones which can occur merely because of their possibility (i.e. captured by the EC rule (4)).

It is assumed in this paper that scenarios are provided by stakeholders. Current work involves integrating ILP and model checking techniques, for instance those presented in [2,11], such that undesirable scenarios are generated automatically using model checking tools. A current assumption of the approach described in this paper is that the scenarios provided to the ILP technique are complete in the sense that events appearing in the sequence are the only events that occur. An area for future research is to relax these assumptions and learn operational requirements from *incomplete* scenarios that satisfy a given specification. Furthermore, system specifications are assumed to be asynchronous. By that we mean that the specification is expected to be satisfied at every position in a path of an LTL model. In goal-oriented requirements engineering approaches, system specifications are usually represented synchronously, i.e. the specification is assumed to hold at certain time points rather than positions. We therefore aim to extend the approach to handle learning synchronous specifications as well as asynchronous ones. Future research also includes extending the specification with other forms of operational requirements such as postconditions which capture additional conditions on fluents that must (not) hold as a consequence of executing event e written as $\Box(e \rightarrow \bigwedge_{1\leqslant i\leqslant n}(\neg)f)$. Finally, a main focus of interest is to incorporate user-defined goals in the learning process to guarantee that the learnt requirements satisfy the stakeholders' goals.

Acknowledgements

This work is funded by CONICET, EPSRC grant PBM C541133/1, UBACyT X020, and ANPCyT PICT grants 32440 and 11738 and the Saudi Arabian Ministry of Higher Education.

References

[1] D. Alrajeh, O. Ray, A. Russo, S. Uchitel, Extracting requirements from scenarios with ILP, in: Proc. of 16th Int. Conference on Inductive Logic Programming, 2006, pp. 63–77.

[2] D. Alrajeh, A. Russo, S. Uchitel, Inferring operational requirements from scenarios and goal models using inductive learning, in: SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ACM, New York, 2006, pp. 29–36.

[3] C. Damas, P. Dupont, B. Lambeau, A. van Lamsweerde, Generating annotated behavior models from end-user scenarios, IEEE Transactions on Software Engineering 31 (12) (2005) 1056–1073. Special Issue on Interaction and State-based Modelling.

[4] C. Damas, B. Lambeau, A. van Lamsweerde, Scenarios, goals, and state machines: a win-win partnership for model synthesis, in: SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, 2006, pp. 197–207.

[5] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: R.A. Kowalski, K. Bowen (Eds.), Proc. of 5th Int. Conference on Logic Programming, 1988, pp. 1070–1080.

[6] D. Giannakopoulou, J. Magee, Fluent model checking for event-based systems, in: ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2003, pp. 257–266.

[7] R.A. Kowalski, M. Sergot, A logic-based calculus of events, New Generation Computing 4 (1) (1986) 67–95.

[8] J. Kramer, J. Magee, M. Sloman, Conic: An integrated approach to distributed computer control systems, IEE Proceedings (Part E) 130 (1983) 1–10.

[9] A. Van Lamsweerde, Goal-oriented requirements engineering: A guided tour, in: Proc. of 5th IEEE Int. Symposium on Requirements Engineering, 2001, pp. 249–263.

[10] A. Van Lamsweerde, L. Willemet, Inferring declarative requirements specifications from operational scenarios, IEEE Transactions on Software Engineering 24 (12) (1998) 1089–1114.

[11] E. Letier, J. Kramer, J. Magee, S. Uchitel, Deriving event-based transitions systems from goal-oriented requirements models, Technical Report 2006/2, Imperial College London, 2005.

[12] E. Letier, A. Van Lamsweerde, Deriving operational software specifications from system goals, in: Proc. of 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2002, pp. 119–128.

[13] D. Lorenzo, Learning non-monotonic logic programs to reason about actions and change, PhD thesis, University of Coruna, 2001.

[14] J. Magee, J. Kramer, Concurrency: State Models and Java Programs, John Wiley and Sons, 1999.

[15] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer, 1992.

[16] R. Miller, M. Shanahan, Some alternative formulations of event calculus, in: Computer Science; Computational Logic; Logic Programming and Beyond, 2002.

[17] S. Moyle, An investigation into theory completion techniques in ILP, PhD thesis, University of Oxford, 2003.

[18] S. Moyle, S. Muggleton, Learning programs in the event calculus, in: Proc. of 7th Int. Workshop on Inductive Logic Programming, 1997.

[19] S.H. Muggleton, Inverse entailment and Progol, New Generation Computing, Special issue on Inductive Logic Programming 13 (3–4) (1995) 245–286.

[20] S.H. Muggleton, C.H. Bryant, Theory completion using inverse entailment, in: Proc. of 10th Int. Conference on Inductive Logic Programming, 2000, pp. 130–146.

[21] S.H. Muggleton, L. De Raedt, Inductive logic programming: Theory and methods, Journal of Logic Programming 19, 20 (1994) 629–679.

[22] R. Otero, Embracing causality in inducing the effects of actions, in: Selected Papers from the 10th Conference of the Spanish Association for Artificial Intelligence (CAEPIA03), in: Lecture Notes in Computer Science, vol. 3040/2004, Springer, Berlin/Heldelberg, 2004, pp. 291–301.

[23] O. Ray, Hybrid abductive-inductive learning, PhD thesis, Imperial College London, 2005.

[24] O. Ray, Using abduction for induction of normal logic programs, in: Proc. of ECAI'06 Workshop on Abduction and Induction in AI and Scientific Modelling, 2006, pp. 28–31.

[25] A. Russo, R. Miller, B. Nuseibeh, J. Kramer, An abductive approach for analysing event-based requirements specifications, in: Proc. of 18th Int. Conference on Logic Programming, 2002, pp. 22–37.

[26] C. Sakama, Induction from answer sets in non-monotonic logic programs, ACM Transactions on Computational Logic 6 (2) (2005) 203–231.

[27] M.P. Shanahan, Solving the Frame Problem, MIT Press, 1997.

[28] C. Stirling, Comparing linear and branching time temporal logics, in: Temporal Logics in Specification, 1987, pp. 1–20.

[29] A. Sutcliffe, N.A.M. Maiden, S. Minocha, D. Manuel, Supporting scenario-based requirements engineering, IEEE Transactions on Software Engineering 24 (1998) 1072–1088.